# Symbolic binary-level code analysis for security. Application to the detection of microarchitectural timing attacks in cryptographic code

Lesly-Ann Daniel

UNIVERSITÉ CÔTE D'AZUR

ÉCOLE DOCTORALE
SCIENCES ET TECHNOLOGIES DE L'INFORMATION ET DE LA COMMUNICATION

# THÈSE DE DOCTORAT

# Analyse Symbolique de Code Binaire pour la Sécurité

## Application à la Détection d'Attaques Microarchitecturales dans les Implémentations Cryptographiques

**Lesly-Ann Daniel**

CEA List, INRIA équipe INDES

**Présentée en vue de l'obtention du grade de docteur en** informatique **d'**Université Côte d'Azur
**Dirigée par** : Tamara Rezk, Directrice de Recherche, *Inria équipe INDES*
**Co-encadrée par** : Sébastien Bardin, Senior Researcher, *CEA List*
**Soutenue le** : 12 Novembre 2021

**Devant le jury, composé de** :
Gilles Barthe, Scientific Director, *Max Planck Institute*
Cristian Cadar, Professor, *Imperial College London*
Aurélien Francillon, Associate Professor, *EURECOM*
Roberto Guanciale, Assistant Professor, *KTH Royal Institute of Technology*
Boris Köpf, Researcher, *Microsoft Research Cambridge*

list cea tech

Inria

UNIVERSITÉ CÔTE D'AZUR

ÉCOLE DOCTORALE STIC

SCIENCES ET TECHNOLOGIES DE L'INFORMATION
ET DE LA COMMUNICATION

# T H È S E

pour obtenir le titre de

**Docteur en Sciences**

de l'Université Côte d'Azur

**Mention : Informatique**

Présentée et soutenue par

Lesly-Ann Daniel

# Symbolic Binary-Level Code Analysis for Security

*Application to the Detection of Microarchitectural Timing Attacks in Cryptographic Code*

Thèse dirigée par Tamara Rezk

préparée au CEA List

soutenue le 12 Novembre 2021

| | | |
|---|---|---|
| *Directeur :* | Tamara Rezk | Directrice de Recherche, *Inria équipe INDES* |
| *Co-encadrant :* | Sébastien Bardin | Senior Researcher, *CEA List* |

**Jury :**

| | | |
|---|---|---|
| *Rapporteurs :* | Gilles Barthe | Scientific Director, *Max Planck Institute* |
| | Cristian Cadar | Professor, *Imperial College London* |
| *Examinateurs :* | Aurélien Francillon | Associate Professor, *EURECOM* |
| | Roberto Guanciale | Assistant Professor, *KTH Royal Institute of Technology* |
| | Boris Köpf | Researcher, *Microsoft Research Cambridge* |

UNIVERSITÉ CÔTE D'AZUR

# *Résumé*

## Analyse Symbolique de Code Binaire pour la Sécurité

Les logiciels informatiques manipulent fréquemment des données secrètes, garantissant généralement leur confidentialité à l'aide de programmes cryptographiques. Dans ce contexte, il est crucial de s'assurer que ces programmes cryptographiques ne peuvent être exploités par un attaquant pour en extraire les données secrètes. Malheureusement, même si les algorithmes cryptographiques sont basés sur des fondations mathématiques solides, leur exécution dans le monde physique produit des effets secondaires pouvant être exploités par un attaquant. En particulier, un attaquant peut exploiter le temps d'exécution d'un programme pour inférer des information sur ses entrées secrètes, ou pour extraire des données secrètes encodées dans la microarchitecture grâce aux *attaques microarchitecturales*. Plus récemment, les attaques *Spectre* ont montré qu'il est aussi possible d'exploiter les optimisations des processeurs — en particulier les *mécanismes de spéculation* — pour extraire des données secrètes.

Dans cette thèse, nous développons des outils d'analyse automatique de programme permettant de vérifier la confidentialité des données secrètes dans les logiciels cryptographiques. En particulier, nous ciblons trois propriétés cruciales pour les programmes cryptographiques : (1) *secret-erasure*, qui s'assure que les données secrètes sont effacées de la mémoire à la fin d'un programme, (2) *constant-time*, qui protège des attaques microarchitecturales, (3) *speculative constant-time*, qui protège contre les attaques Spectre. Ces propriétés ont en commun deux caractéristiques qui les rendent difficiles à analyser. Premièrement, il s'agit de propriétés de *paires* d'exécution (c'est-à-dire de 2-hypersûreté), ce qui les rend incompatibles avec les outils de vérification habituels (conçus pour la sûreté). Deuxièmement, elles ne sont généralement pas préservées par les compilateurs.

Dans cette thèse, notre objectif est de concevoir des outils d'analyse symbolique automatiques, modélisant des paires d'exécution, fonctionnant au niveau du code binaire, incluant les mécanismes de spéculation des processeurs, et passant à l'échelle sur des logiciels cryptographiques existants. Nos analyses sont basées sur l'exécution symbolique relationnelle — une adaptation de l'exécution symbolique à la 2-hypersûreté — que nous complétons avec des optimisations permettant de : (1) la rendre efficace au niveau binaire, et (2) de modéliser de manière efficace la sémantique spéculative des programmes. Nous proposons deux outils *open source* : Binsec/Rel, un outil pour la recherche de bugs et la vérification bornée de constant-time et secret-erasure ; et Binsec/Haunted, un outil pour détecter des vulnérabilités aux attaques Spectre. Notre évaluation expérimentale montre que nos optimisations permettent une amélioration drastique des performances en comparaison aux approches antérieures : elles diminuent le temps d'analyse, trouvent plus de bugs, et permettent la vérification de primitives cryptographiques jusqu'alors hors de portée des approches antérieures.

Nous analysons, grâce à nos outils, un large éventail de primitives cryptographiques et de fonctions utilitaires de bibliothèques open source (telles que Libsodium, OpenSSL, BearSSL and HACL*) pour constant-time (338 binaires), secret-erasure (680 binaires), et Spectre (45 binaires). Au cours de ces travaux, nous découvrons quelques bugs nouveaux, ainsi que des faiblesses dans certaines protections logicielles.

**Mots clés :** analyse de binaire, méthodes formelles, vérification logicielle, exécution symbolique, constant-time, secret-erasure, Spectre.

UNIVERSITÉ CÔTE D'AZUR

# *Abstract*

**Symbolic Binary-Level Code Analysis for Security**

Programs commonly perform computations involving secret data, relying on cryptographic code to guarantee their confidentiality. In this context, it is crucial to ensure that cryptographic code cannot be exploited by an attacker to leak secret data. Unfortunately, even if cryptogtaphic algorithms are based upon secure mathematical foundations, their execution in the physical world can produce side-effects that can be exploited to recover secrets. In particular, an attacker can exploit the execution time of a program to leak secret data, or use timing to recover secrets encoded in the microarchitecture using *microarchitectural timing attacks*. More recently, *Spectre attacks* showed that it is also possible to exploit processor optimizations—in particular *speculation mechanisms*—to leak secret data.

In this thesis, we develop automated program analyses for checking confidentiality of secret data in cryptographic code. In particular, we target three crucial properties of cryptographic implementations: (1) *secret-erasure*, which ensures that secret data are erased from memory at the end of the program; (2) *constant-time*, which protects against microarchitectural timing attacks; (3) *speculative constant-time*, which protects against Spectre attacks. These properties have two characteristics in common that make them challenging to analyze. First, they are properties of *pairs* of traces (namely 2-hypersafety), which makes them incompatible with the standard verification framework (designed for safety properties). Second, they are not always preserved by compilers.

Our goal in this thesis is to design automatic symbolic analyses for pairs of traces, operating at binary-level, including processor speculations, and that scale on real-world cryptographic code. Our analyses are built on top of relational symbolic execution—the adaptation of symbolic execution to 2-hypersafety—that we complement with dedicated optimizations: (1) for efficient relational symbolic execution at binary-level, (2) for modeling efficiently the speculative semantics of programs. We implement our analyses into two *open-source* tools: Binsec/Rel, a tool for bug-finding and bounded-verification of constant-time and secret-erasure; and Binsec/Haunted, a tool for detecting vulnerabilities to Spectre attacks. Our experimental evaluation shows that our optimizations drastically improve performance over prior approaches, allowing for faster analysis, finding more bugs, and enabling bounded-verification on real-world cryptographic primitives whereas prior approaches times-out.

Using our tools, we analyze a wide range of cryptographic primitives and utility functions from open-source libraries (including Libsodium, OpenSSL, BearSSL and HACL*) for constant-time (338 binaries), secret-erasure (680 binaries), and Spectre (45 binaries). Along the way, we discover a few new bugs, as well as weaknesses in standard protections schemes.

**Keywords:** binary analysis, formal methods, software verification, symbolic execution, constant-time, secret-erasure, Spectre.

# *Acknowledgements*

First and foremost, I would like to thank my advisors Sébastien and Tamara. Thank you for believing in me and for encouraging me to try ambitious things. Thank you also for all your valuable advices and careful proofreading which greatly helped me progress and improve my work. I really enjoyed working with you for these three years.

My sincere thanks also goes to Gilles Barthe, Cristian Cadar, Aurélien Francillon, Roberto Guanciale, and Boris Köpf who agreed to be part of my jury and in particular, Gilles Barthe and Cristian Cadar for reviewing my thesis.

I also thank Frank Piessens for his invitation to collaborate on an exciting project and for accepting me as a postdoc in his team.

I thank the members of the INDES team at Inria for welcoming me during a couple of month when I was writing my thesis, which made my writing experience much more enjoyable. I also thank Christopher Hauser for welcoming me in his team at University of Southern California.

I thank the CEA, in particular the LSL, for providing such a great work environment. Thank you to all my colleagues, you are the best colleagues one could wish for! I really enjoyed the time I spent with you, including the coffee breaks, the after work beers, the picnics, all the great discussions, etc.

A special thank goes to all my fellow PhD students, for all the support you provided. I am glad that I have shared this experience with you.

I am also glad that I had the chance to present my work to a more general audience, once with Florent and our PSES dream team, and once with Myriam; and I thank all the colleagues who helped us improve our presentations.

I am also grateful to the many people that I met during my journey in academia, who made me discover new cultures and made me feel included when I was far from home, in particular Sander and Sima.

Je suis infiniment reconnaissante pour toutes les personnes exceptionnelles que j'ai rencontrées pendant ces trois ans et pour tous les moments passés ensemble. Je cite en vrac : les animateurs de lsl-misc, les randonnées de la Bühler voyage company, Eugénie pour les après-midis guitare et les randonnées, les coureurs du LSL, toute la fabuleuse team Mississippi, la pause café dissidente et ses intenses discussions, les nombreuses discussions Matrix, les amis de Morzine, les amis de PSL, les soirées jeux, les soirées tout court. Merci aussi à tous ceux que je n'ai pas cités mais qui m'ont accompagné pendant ces trois ans.

Je remercie Clément, Romain et Simon qui sont venus de loin pour assister à ma soutenance. Je remercie aussi ma mère, mon père, ma soeur pour avoir toujours cru en moi. Enfin, mes plus profonds remerciements sont pour Olivier. Merci pour ta patience, pour ton support inconditionnel, et pour m'avoir suivie dans toutes mes folles aventures.

# Contents

# List of Symbols

**DBA syntax.** DBA syntax, for modeling low-level programs, is defined in Figure 2.3 at page 26.

**Mathematical notations.**

| | |
|---|---|
| $\wp$ | Powerset |
| $\mathbb{N}$ | Natural numbers |
| $\lvert S \rvert$ | Size of $S$ |

**Concrete notations.**

| | |
|---|---|
| $\mathtt{bv}, \mathtt{bv'} \in BV_n$ | Bitvectors of size $n$ |
| $l, l' \in Loc$ | Program locations |
| $\mathtt{v} \in \mathcal{V}$ | Program variables |
| $Inst$ | Set of program instructions |
| $\mathbb{P} : Loc \to Inst$ | Program |
| $\mathbb{P}[l]$ | Instruction at location $l$ in program $\mathbb{P}$ |
| $r : \mathcal{V} \to BV_{32}$ | Concrete register map |
| $m : BV_{32} \to BV_8$ | Concrete memory |
| $c \triangleq (l, m, r)$ | Concrete configuration |
| $c \underset{t}{\to} c'$ | Concrete evaluation of instructions producing leakage $t$ |
| $c\ e \vdash_t \mathtt{bv}$ | Concrete evaluation of expression $e$ producing leakage $t$ |

**Symbolic notations.**

| | |
|---|---|
| $\varphi, \phi, \psi, \dots \in \Phi$ | Symbolic expressions in QF_ABV logic |
| $\beta, \beta' \in \mathcal{B}l$ | Symbolic boolean expressions |
| $bv, bv' \in \mathcal{B}v_n$ | Symbolic bitvectors of size $n$ |
| $\vDash \pi$ (resp. $\nvDash \pi$) | Satisfiability (resp. unsatisfiability) of formula $\pi$ |
| $\vDash_{\text{SMT}} \pi$ (resp. $\nvDash_{\text{SMT}} \pi$) | Check (un-)satisfiability of formula $\pi$ with an SMT solver |
| $\widehat{\varphi}, \widehat{\phi}, \widehat{\beta}, \widehat{\psi}, \dots \in \mathbf{\Phi}$ | Lifting of set $S$ (resp. function $f$) to relational expressions |
| $\langle \varphi \rangle$ | Simple relational expression |
| $\langle \varphi_l \,\vert\, \varphi_r \rangle$ | Pair of relational expressions |
| $\widehat{\varphi}_{\vert l}$ resp. $\widehat{\varphi}_{\vert r}$ | Left (resp. right) projection of $\widehat{\varphi}$ |
| $\rho : \mathcal{V} \to \mathbf{\Phi}$ | Symbolic register map |
| $\widehat{\mu} \in (Array\ \mathcal{B}v_{32}\ \mathcal{B}v_8)^2$ | Symbolic (relational) memory |
| $\pi \in \Phi$ | Path predicate |
| $s \triangleq \big(l, \rho, \mu, \pi\big)$ | Symbolic configuration |
| $s \rightsquigarrow s'$ | Symbolic evaluation of instructions |
| $\big(\rho, \widehat{\mu}, \pi\big)\ e \vdash \widehat{\varphi}$ | Symbolic evaluation of expression $e$ |
| $c \approx^M_p s$ | Concretization relations of the $p$-side of $s$ with model $M$ (see Definition 10) |
| $\mathcal{M}(\widehat{\varphi}, \pi)$ | Set of concrete values that $\widehat{\varphi}$ can take to satisfy $\pi$ |

**Security evaluation.**

| | |
|---|---|
| $\mathcal{V}_l$ resp. $\mathcal{V}_h$ | Set of low (resp. high) variables |
| $\mathcal{A}_l$ resp. $\mathcal{A}_h$ | Set of low (resp. high) addresses |
| $c \simeq_l c'$ | Low equivalence of states (see Definition 11) |
| $secleak(\widehat{\varphi}, \pi)$ | Check if $\widehat{\varphi}$ can be leaked securely under path predicate $\pi$ (see Section 4.4.1.1) |

**Notations for Haunted RelSE.**

| | |
|---|---|
| $\widetilde{\delta} \in \mathbb{N}$ | Current depth of symbolic execution |
| $\Delta \in \mathbb{N}$ | Maximum speculation depth |
| $(\widehat{\varphi}, \delta)$ | Expression $\widehat{\varphi}$ with a retirement depth $\delta$ |
| $\widehat{\varphi}^{\delta}$ | Expression $\widehat{\varphi}$ with a memory-dependency depth $\delta$ |
| $\widetilde{\pi} \in \wp(\mathcal{B}l \times \mathbb{N})$ | Speculative path predicate |
| $\widetilde{\lambda} \in \wp(\mathcal{B}l \times \mathbb{N})$ | Set of transient loads |

**Experimental evaluation.**

| | |
|---|---|
| Time | Execution time of the analysis |
| Inst | Number of instructions of a program |
| $\mathrm{I}_{\texttt{x86}}$ | Number of static instructions explored by the analysis |
| $\mathrm{I}_{\mathrm{unr}}$ | Number of unrolled instructions explored by the analysis |
| Paths | Number of paths explored |
| $\mathrm{Q}_{\mathrm{expl}}$ | Number of exploration queries sent to the solver |
| $\mathrm{Q}_{\mathrm{insec}}$ | Number of insecurity queries sent to the solver |
| $\mathrm{Q}_{\mathrm{tot}}$ | Total number of queries sent to the solver |
| Status | Status of the analysis set to ✓ for secure (exhaustive exploration), ✗ for insecure or ♜ for timeout |
| 🐞 | Number of bugs |

# Part I

# Introduction

# Chapter 1

# Introduction

## 1.1 Context

**Protecting secret data used in programs.** In our everyday life, we commonly entrust programs with confidential data, or *secret data*, for instance in online banking, online tax returns, encrypted communications, etc. To preserve the security of these systems, it is crucial to ensure that these programs cannot be exploited by an attacker to leak user's secrets. Take for instance the password checker in Listing 1.1 which checks whether a user's attempt—possibly controlled by an attacker—corresponds to a secret password. The goal of an attacker is to recover some information about the secret `password`. A first step to protect secret data is to make sure that they are erased from the memory after the execution of critical code, as illustrated in Listing 1.1. This policy, called *secret-erasure* [73], limits the time secret data reside in memory and protects them against subsequent memory disclosure vulnerabilities.

```
1 bool check_password(char attempt[LEN]) {
2     char password[LEN];
3     get_password(password);
4     for (int i = 0; i < LEN; i++)
5         if (password[i] != attempt[i])
6             return false;
7     clear_password(password);
8     return true;
9 }
```

Listing 1.1 – Example of password checker.

Unfortunately, ensuring that secret data do not leak to observable program outputs and guaranteeing the absence of memory disclosure vulnerabilities is not sufficient for ensuring the confidentiality of secret data. Indeed, when software are executed on hardware, they produce measurable physical effects that can leak information about their secret input, for instance via execution time [156], power consumption [157], electromagnetic emissions [209], radio signals [64], noise [116], etc. This class of attacks, pioneered by Paul Kocher in 1996 [156], are called *side-channel attacks*.

*Timing attacks* are a special case of side-channel attacks that exploit the execution time of a system to leak secrets. First timing attacks exploited measurable differences in the execution time of a victim's program happening when the sequence of executed instructions depends on secret data. Consider again the password checker code in Listing 1.1: the number of iterations of the loop—and hence the execution time of the program—depends on the number of correct initial characters of `attempt`. An attacker can make several guesses and, by measuring timing variations, estimate the

number of correct initial characters, eventually recovering the secret `password` one character at a time.

Timing attacks are particularly devastating as they do not necessitate special equipment and can be run remotely by an attacker to recover full secret keys [51, 52, 50]. In a less obvious way, timing variations can also depend on the sequence of memory accesses of the victim, because of cache hits and misses. An attacker can abuse this mechanism and recover secrets via *cache timing attacks* [39]. More generally, computations leave traces in the hidden state of the microarchitecture—i.e. the way an architecture is implemented in a processor. If a victim leaks secrets to the microarchitectural state, these secrets can be recovered by an attacker in a shared physical environment, like a virtual machine or a cloud service, via *microarchitectural timing attacks*. Microarchitectural channels include for instance cache state [39, 195, 131, 200, 2, 123], branch predictors [3, 104, 103], port contention [5], AVX [225], etc.

A solution to protect software against timing attacks is to adopt the *constant-time* [30] programming discipline (a.k.a. constant-time policy). Constant-time programming consists in writing a program in such a way that its *control-flow* and the *address of memory accesses* do not depend on secrets[1]. For instance, the password checker given in Listing 1.1 violates the constant-time policy because the condition at line 5 depends on the value of the secret `password`. A constant-time implementation, such as the one given in Listing 1.2, should not branch on the secret and always execute the same number of loop iterations. Constant-time is the most effective countermeasure for protecting programs against timing attacks [30] and is already used in many cryptographic implementations [272, 36, 41].

```
1  bool check_password(char attempt[LEN]) {
2      char password[LEN];
3      get_password(password);
4      bool good = true;
5      for(i = 0; i < LEN; ++i) {
6          good &= password[i] == attempt[i];
7      }
8      clear_password(password);
9      return good;
10 }
```

Listing 1.2 – Constant-time version of the program in Listing 1.1 (assuming the comparison `password[i] == attempt[i]` is implemented using branchless logic).

Since 2018, a new class of microarchitectural attacks, called *transient execution attacks* [155, 172], has been made public, opening new opportunities for attacks. These attacks exploit processor optimizations in order to leak secrets into the microarchitectural state. In particular, Spectre [155] attacks take advantage of out-of-order execution and speculation mechanisms in order to trigger invalid sequence of instructions—called *transient executions*—that encode secrets into the microarchitectural state—for instance the cache. Thus far, there are four variants of Spectre [65] according to the speculation mechanism exploited. In this thesis, we focus on *Spectre-PHT* and *Spectre-STL* respectively exploiting the conditional branch predictor and the memory disambiguation mechanism.

1. Some versions of constant-time also require that the operands of variable-time instructions [125, 14] do not depend on secret.

Constant-time programming is not sufficient to protect against Spectre attacks because it does not consider additional behavior introduced by transient executions. Fortunately, an adaptation of constant-time, called *speculative constant-time* [67], has been recently proposed to encompass these attacks. The idea behind speculative constant-time is to enforce constant-time on the speculative semantics of the program—comprising both sequential and transient executions.

*The general context of this thesis is to develop program analysis techniques to help enforce secret-erasure, constant-time and speculative constant-time, with a particular focus on cryptographic code.*

**Program analysis for security.** Formal methods [75] for program analysis have been developed to verify (or refute) that existing programs satisfy given properties, for instance that critical systems do not have runtime errors or unexpected behaviors. *Sound program analysis*, which *over-approximate* the semantics of a program (such as abstract interpretation [81]) offer strong guarantees that a program satisfies a property. However, they cannot precisely find bugs and can have high numbers of false alarms when applied to large systems that are not initially designed for formal verification and are often limited to small critical systems. On the contrary, *bug-finding* techniques, which *under-approximate* the semantics of a program (e.g. symbolic execution [151] or bounded model-checking [44]) cannot prove that a program satisfy a property but can precisely identify violations and are much easier to deploy on real-wold systems.

In the context of security, many program analysis techniques—especially for bug-finding [150]—focus on detecting exploitable violations of safety properties such as buffer overflows, use-after-free, race conditions, etc. and numerous efficient tools have been developed for analyzing safety properties [58, 121, 152, 136, 146, 82, 19]. *Safety properties* [12] are properties of *individual execution traces*. Informally, safety properties state that "bad things" do not happen during the execution of a program [12], and a counterexample for a safety property is a finite execution trace exhibiting this "bad thing". For instance, use-after-free is a safety property where the "bad thing" is dereferencing a dangling pointer, and a counterexample is an execution trace of a program in which a dangling pointer is dereferenced.

However many important security properties—e.g. protecting user's secrets from an attacker—cannot be expressed as properties of individual traces but are expressed as properties of *sets of traces*—i.e. they are *hyperproperties* [77]. In particular, *information flow properties*, which regulate the leakage of information from secret input of a program to public output, relate two execution traces—i.e. are *2-hypersafety properties* [77]. These properties are necessary to express security policies of cryptographic implementations such as *constant-time*, cache side-channel freedom, *secret-erasure*, or the absence of transient execution attacks. Consequently, *it is crucial to develop efficient automated verification tools to verify 2-hypersafety—or to find bugs.* While sound analysis have been developed for hyperproperties (e.g. type systems [218], logics [76], model checking [110, 109], etc.), bug-finding techniques for hyperproperties are still behind [150].

*In this thesis, we try to bridge this gap by developing techniques for both bounded-verification and bug-finding of 2-hypersafety. Our techniques can scale on real-world program to find bugs; and can also provide strong guarantees when a program is secure.*

**Binary-level analysis** Unfortunately, 2-hypersafety properties are generally not preserved by compilers [231, 101, 43]. In Listing 1.1 for instance, *secret-erasure* is enforced at source-level by erasing secret data from the memory at the end of the program with a function `clear_password`, which could be `memset(password, 0, LEN)`.

This write operation does not influence the output of the program and therefore, can be optimized-away by the dead-store-elimination pass of the compiler [269, 43, 101, 83], hence violating the secret-erasure policy in the executable code. Similarly, reasoning about microarchitectural timing attacks requires to precisely reason about conditional branches or store and load operations, yet these instructions can be added at compiled time and are not always visible at source level [231, 43].

A first approach to solve this challenge is to use a formally verified compiler such as CompCert [168] which preserves the properties through compilation. Along this line, constant-time preservation was added to CompCert very recently [31]. However CompCert is not as optimizing as mainstream compilers like `clang` or `gcc` and for this reason, it is rarely used beyond safety-critical developments. Another example is the Jasmin [7] ecosystem, for developing cryptographic software in a specialized low-level language, coming with a formally verified and optimizing compiler that preserves constant-time. However, this approach requires to re-implement cryptographic primitives and does not apply to existing programs.

*In this thesis, we adopt a different approach, applicable to existing programs: we analyze these properties directly at binary level on existing cryptographic codes.*

**Symbolic execution for 2-hypersafety.** A technique that scales well on binary code and that naturally comes into play for bug-finding and bounded-verification is *symbolic execution* (SE) [121, 63]. While it has proven very successful for standard safety properties [48], its direct adaptation to 2-hypersafety through (variants of) self-composition suffers from a scalability issue [24, 98, 182]. Some recent approaches achieve better scaling, but at the cost of sacrificing either bounded-verification [254, 238]—hence sacrificing the guarantees they can offer on secure programs—or bug-finding [49]—making them of minor interest when the program cannot be proven secure.

## 1.2 Goal and challenges.

Our goal is to adapt binary-level symbolic execution for bug-finding and bounded-verification of information-flow properties, in order to scale on real-world cryptographic software. In particular, we target three crucial properties of cryptographic code, namely *secret-erasure*, *constant-time* and *speculative constant-time*. This goal can be divided into two main objectives. First, we need to adapt symbolic execution for reasoning about 2-hypersafety at binary-level. Second, we need to reason about program behaviors (i.e. microarchitectural state and transient executions) that are not visible at the architectural level.

### 1.2.1 Binary analysis for 2-hypersafety

Designing automated verification tools for 2-hypersafety at binary-level is challenging for the following reasons:

**C1** Common verification methods designed for safety do not directly apply to 2-hypersafety. While the problem of verifying 2-hypersafety properties can be reduced to verifying safety properties on a transform program by *self-composition* [33], it is inefficient in practice [241]. Consequently, there is a need for dedicated techniques to efficiently analyze properties relating pairs of traces;

**C2** It is notoriously difficult to adapt formal methods to binary-level because of the lack of structure information (data and control) and the explicit representation of the memory as a large array of bytes [97, 23].

Instead of reducing 2-hypersafety to safety via self-composition and use off-the-shelf verification tools, a recent promising approach is to implement dedicated analyzers to model *pairs* of executions efficiently [38, 266, 18, 105]. The idea of analyzing pairs of executions in a single symbolic execution instance while maximizing sharing between these pairs of executions originates from back-to-back testing and has first been coined as *ShadowSE* [62]. This idea has been reused later in the context of 2-hypersafety verification and called *relational symbolic execution* (RelSE) [105].

**Proposal.** In a nutshell, we propose solving challenges **C1** and **C2** by adapting relational symbolic execution to binary analysis. However, we show in this thesis that a direct adaptation of RelSE *does not scale in the context of binary-level analysis* to analyze constant-time on real cryptographic implementations. This is because of the representation of the memory as large symbolic array which cannot be shared between pairs of executions. Therefore, we propose dedicated optimizations for *binary-level RelSE*, offering fine-grained information flow tracking in the memory. These optimizations allow for better sharing at binary-level and enhanced tracking of secret dependencies, leading to a reduction of the number of queries sent to the constraint solver.

### 1.2.2 Reasoning about microarchitectural security

Reasoning about microarchitectural attacks is fundamentally different from reasoning about standard properties because it requires to take into account program behavior that is not visible at the architectural level:

**C3** The source of leakage depends on the details of the microarchitecture which is often not publicly accessible [179] and too complex to model precisely, e.g. cache state [39, 195, 131, 200, 2, 123], branch predictors [3, 104, 103], port contention [5], AVX [225], etc.;

**C4** Reasoning about Spectre attacks requires to take into account the *speculative semantics* of programs [67], with out-of-order execution and speculation mechanisms. Modeling the new behavior introduced by transient executions can quickly degrade the precision or the performance of the analysis.

**Proposal.** To address challenge **C3**, we target a property called *constant-time* which offers an abstraction to reason about microarchitectural side-channels without modeling the details of the microarchitecture, and guarantees the absence of timing leaks [30]. Moreover, building on a parametric leakage semantics from prior work [35], we generalize our analysis by making it parametric in the leakage model. This allows us to target secret-erasure in addition to different variations or constant-time.

Finally, to model Spectre attacks we extend our analysis to *speculative constant-time* [67] which is a property analogous to constant-time that takes speculations into account while abstracting away intricate details of the microarchitecture. However we show that modeling explicitly in symbolic execution all additional transient executions introduced by the speculative semantics can quickly lead to path explosion. Our key idea to address challenge **C2** is to adapt relational symbolic execution to model both transient executions and sequential executions *at the same time*, which we call *Haunted RelSE*.

## 1.3 Contributions

In this thesis, we tackle the problem of designing *efficient symbolic analyzers for verifying information flow properties at binary-level*. We restrict to a subset of information flow properties, relating traces following the same path, that includes crucial properties of cryptographic implementations such as constant-time (cf. Chapter 4), secret-erasure (cf. Chapter 5), and speculative constant-time (cf. Chapter 6).

### 1.3.1 Primary contributions overview

**Technical contributions.** Our main challenge is to design analyzers that scale on real-world cryptographic code. Our technical contribution for addressing this challenge is to *design dedicated optimizations* tailored to the problem at hand:

— We propose dedicated optimizations for relational symbolic execution at binary level, named *binary-level RelSE* (Chapter 4). Considering that, in binary-level RelSE, the explicit representation of the memory prevents sharing between pairs of executions, our key technical insight is to propose a shared memory representation, based on *read-over-write* [107], in order to improve sharing in the memory, track information flow, and reduce the number of queries sent to the solver;

— We propose dedicated optimization for modeling speculative semantics in symbolic analyses, named *Haunted RelSE* (Chapter 6). Considering that modeling speculative semantics in RelSE quickly leads to path explosion, our key technical insight is to model sequential paths and transient paths *at the same time* using a combination of sound path pruning and logical encoding of the remaining paths.

We argue that proposing both bug-finding and bounded-verification in one tool is a major advantage because binary-level tools usually have a steep learning curve and a developer who wants to analyze their code only has to master a single tool. Consequently, one of our concerns is to design techniques that *scale on real-world cryptographic code without sacrificing bug-finding nor bounded-verification*. For this, we *formally prove* our analyses correct for bug-finding (cf. Theorem 2) and bounded-verification (cf. Theorem 4). Moreover, the optimizations that we propose do not approximate the semantics of programs—which we formally prove for Haunted RelSE (cf. Theorem 6).

**Implementation of two symbolic analyzers.** On the practical side, we implemented our techniques into two tools that we open-sourced on github:

— Binsec/Rel [85]: the first efficient automatic analyzer for bug-finding and bounded-verification of constant-time and secret-erasure at binary-level;

— Binsec/Haunted [86]: the adaptation of Binsec/Rel to efficiently model the speculative semantics of programs and find Spectre-PHT and Spectre-STL vulnerabilities.

These tools are compiler-agnostic, target 32-bit x86 and ARM architectures and do not require source code. Both tools and their underlying techniques have been evaluated on cryptogtaphic binary codes, achieving better performance than state-of-the-art approaches. In our experiments, Binsec/Rel is 715× faster than standard RelSE, achieving exhaustive bounded-verification on large programs—e.g. 17 minutes for an implementation of Curve25519-donna [40] with 10 millions unrolled instructions—whereas the latter times out. Binsec/Haunted can exhaustively analyze code up to 5k static instructions for the Spectre-PHT and is faster than state-of-the-art tools

KLEESpectre [252] and Pitchfork [67]. For Spectre-STL, it can exhaustively analyze codes up to 100 instructions and find vulnerabilities in codes up to 6k instructions.

**Application to cryptographic code.** On a more concrete perspective, these tools have been applied to perform extensive analysis of cryptographic primitives from well known libraries (Libsodium [41], OpenSSL [192], BearSSL [207] and HACL* [272]); and to automate and extend prior manual study on the preservation of constant-time [231] and secret-erasure [269]. In total, we analyze:

— 338 binaries for constant-time, including 18 binaries from cryptographic primitives and 320 configuration from utility functions;

— 680 binaries for secret-erasure (i.e. 17 scrubbing functions in 40 compilers configurations);

— 45 binaries for Spectre-PHT and Spectre-STL, including 13 configuration from 5 real-world cryptographic primitives and 13 new litmus tests for Spectre-STL that we propose.

Interestingly, using BINSEC/REL and BINSEC/HAUNTED we were able to detect new violations:

— We discovered that `gcc -O0` and backend passes of `clang` may introduce constant-time violations that cannot be detected at LLVM level, which shows the importance of reasoning at binary-level;

— We discovered that scrubbing functions implemented using volatile function pointers can introduce additional register spilling that might break secret-erasure when compiled with `gcc -O2` and `gcc -O3`;

— We discovered that index-masking, a well-known defense against the most prominent variant of Spectre, may introduce vulnerabilities with regard to another, less well-known, variant of Spectre;

— We also found that PIC options [112] from the `gcc` compiler may introduce Spectre violations.

*Overall, our contribution is twofold. First, we show that, with appropriate optimizations, binary-level relational symbolic execution scales on cryptographic code, even when considering speculative execution. Second, we propose flexible tools for analyzing several properties and apply them on existing cryptographic codes.*

### 1.3.2 Secondary contributions

**External repositories.** During this thesis, we developed and open-sourced the following tools:

— BINSEC/REL*: a symbolic binary analyzer for constant-time and secret-erasure* [85] has been released at `https://github.com/binsec/haunted` and the benchmarks at `https://github.com/binsec/rel_bench`;

— BINSEC/HAUNTED*: a symbolic binary analyzer to detect Spectre attacks* [86] has been released at `https://github.com/binsec/haunted` and the benchmarks at `https://github.com/binsec/haunted_bench`;

— We also developed an easily extensible framework for studying preservation of secret-erasure by compilers [84], available at `https://github.com/binsec/rel_bench/tree/main/src/secret-erasure`;

— We propose a new set of litmus tests in order to evaluate analysis tools for Spectre-STL, available at https://github.com/binsec/haunted_bench/blob/master/src/litmus-stl/programs/spectrev4.c.

**Papers.** The work presented in Chapters 4 and 6 has been *published* in the following papers:

— *Binsec/Rel: Efficient Relational Symbolic Execution for Constant-Time at Binary-Level* [87], Lesly-Ann Daniel, Sébastien Bardin, Tamara Rezk. Published in the proceedings of IEEE Symposium on Security and Privacy (SP) in 2020;

— *Hunting the Haunter—Efficient Relational Symbolic Execution for Spectre with Haunted RelSE* [88], Lesly-Ann Daniel, Sébastien Bardin, Tamara Rezk. Published in the proceedings of the Network and Distributed System Security Symposium (NDSS), 2021.

The work presented in Chapter 5 have been *submitted* to:

— *Binsec/Rel: Symbolic Binary Analyzer for Security with Applications to Constant-Time and Secret-Erasure*, Lesly-Ann Daniel, Sébastien Bardin, Tamara Rezk. Sumbitted to ACM Transactions on Privacy and Security (TOPS), 2021. *(Under review).*

Finally, the work in Appendices B and C.1 has been adapted from a presentation to the LASER Workshop 2021 and *submitted* to the post-workshop proceedings:

— Reflections on the Experimental Evaluation of a Binary-Level Symbolic Analyzer for Spectre, Lesly-Ann Daniel, Sébastien Bardin, Tamara Rezk. Sumbitted to the proceedings of Learning from Authoritative Security Experiments Results (LASER), 2021. *(Under review).*

**Additional work.** During this thesis, we also worked on the design and formal analysis of hardware countermeasures enabling secure speculation. Along this line, we propose a microarchitectural semantics that generalizes speculation mechanisms; design a hardware monitor that enables secure speculation; and formally prove its security guarantees. This work has been done in collaboration with Márton Bognár, Job Noorman, and Frank Piessens and is still in progress.

## 1.4 Outline

This thesis starts with two background chapters:

— Chapter 2 introduces key concepts in automated program analysis, stressing the difference between analysis that over-approximate the semantics of programs (i.e. sound analyses) and analyses that under-approximate the semantics of programs (i.e. bug-finding techniques). It introduces a particular program analysis technique called *symbolic execution* that can be used for both sound analysis and bug-finding and which we use in this thesis. It also introduces binary analysis pointing out why it is needed and how does it differ from source code analysis, presents the BINSEC binary analysis platform in which we implement our tools. Finally, it introduces a low level language called DBA that we use in this thesis to model low-level programs and a symbolic execution for this low-level language;

— Chapter 3 introduces the properties that we analyze in this thesis. It starts by defining information flow analysis and noninterference, pointing out that these properties are not regular safety properties but 2-hypersafety properties. It presents microarchitectural timing attacks and a common software-based mitigation called constant-time that we analyze in this thesis. Finally, it presents transient execution attacks—with a particular focus on Spectre-PHT and Spectre-STL, the two variants of Spectre addressed in this thesis—and introduces the speculative constant-time policy which is the adaptation of constant-time that additionally encompasses Spectre attacks.

The main contributions presented in this thesis are developed in three contribution chapters:

— Chapter 4 presents a technique for efficient constant-time analysis at binary-level based on relational symbolic execution, enhanced with dedicated optimization. It proposes BINSEC/REL, the first efficient tool for bug-finding and bounded-verification of constant-time programs and performs extensive experiments on a set of 338 cryptographic binaries, showing the advantage of the technique. It also automate a prior manual study on the preservation of constant-time by compilers and, interestingly, finds new vulnerabilities introduced by backend passed of `clang` and out-of-reach of LLVM verification tools;

— Chapter 5 generalizes binary-level RelSE presented in Chapter 4, to a subset of information flow properties encompassing constant-time and secret-erasure. It extends BINSEC/REL to verify secret-erasure. Finally, it proposes an easily extensible framework to verify various enforcement mechanisms for secret-erasure compiled with multiple compilers and, using this framework, automates a prior manual study on preservation secret-erasure by compilers. Interestingly, it shows that enforcement of secret-erasure using volatile function pointers may introduce additional register spilling, hence violating secret-erasure;

— Chapter 6 extends binary-level RelSE presented in Chapter 4 for the detection of Spectre attacks and proposes optimizations, called Haunted RelSE, to efficiently explore the speculative behavior of programs. It proposes a prototype tool for Haunted RelSE, called BINSEC/HAUNTED, and compare it against standard approaches and against two state-of-the-art tools on a set of small test cases and on real cryptographic implementations. These experiments show that Haunted RelSE can find more violations and scales better than state-of-the-art techniques and tools. Finally, it reports that a standard defense for Spectre-PHT (i.e. index-masking), options to compile position independent code in `gcc` may introduce Spectre-STL violations.

Finally, we conclude and give some perspectives for future work in Chapter 7.

**How to read.** Each contribution chapter can be read independently from the others. Chapters 5 and 6 build on the theoretical foundations introduced in Chapter 4 and are therefore easier to read after reading this chapter. Still, they are self-contained and the key concepts and notations are reminded using reminder boxes. Additionally, all notations are summarized in the list of symbols at page xi.

**Reminder**

This reminder box summarizes content from a previous chapter.

# Part II

# Background

# Chapter 2

# Automated Program Analysis

**Chapter overview**

This chapter introduces technical background on program analysis.

— Section 2.1 gives an overview of program analysis, in particular on the difference between *sound analyses* and *bug-finding* techniques;

— Section 2.2 introduces a particular program analysis technique called *symbolic execution*, which we use in this thesis;

— Section 2.3 introduces binary analysis and its challenges and presents the BINSEC analysis platform in which we implement our analyzers. Finally, it introduces a low-level language called DBA with its formal semantics, and presents a symbolic execution for this low-level language.

## 2.1 Overview of program analysis

Formal methods [75] are mathematically-based techniques to reason on properties of programs. A program $\mathbb{P}$ is defined as the a set of possible behaviors [1] permitted by the semantics of its source code. A *property*, usually specified as a mathematical formula on the semantics of programs, defines a set $\mathcal{P}$ or acceptable program behaviors.

**Definition 1** ($\mathbb{P} \vDash \mathcal{P}$)**.** *A program* $\mathbb{P}$ *satisfies a property* $\mathcal{P}$*, written* $\mathbb{P} \vDash \mathcal{P}$*, if and only if* $\mathbb{P} \subseteq \mathcal{P}$*.*

**Definition 2** ($\mathbb{P} \nvDash \mathcal{P}$)**.** *Conversely, a program* $\mathbb{P}$ *violates a property* $\mathcal{P}$*, written* $\mathbb{P} \nvDash \mathcal{P}$*, if and only if* $\exists c \in \mathbb{P}.\ c \notin \mathcal{P}$*. A particular program behavior* $c$ *that violates the property is called a* counterexample *or a* bug*.*

Automated program analyses try to automatically decide whether a program $\mathbb{P}$ satisfies a property $\mathcal{P}$ or not. To do this, they compute a set of behaviors $\mathcal{A}$ modeling the real program behaviors, for which they are able to decide whether $\mathcal{A} \vDash \mathcal{P}$ or $\mathcal{A} \nvDash \mathcal{P}$.

Unfortunately, Rice theorem states that "all non-trivial semantic properties of programs are undecidable". Consequently, automatic static analyses cannot always correctly decide meaningful properties when they consider non-trivial behaviors of $\mathbb{P}$. They have to *approximate* the set of real program behaviors. Two main approaches can be adopted for automated program analysis, offering complementary tradeoffs:

1. We purposely leave the definition of behavior abstract for now and refine it in Section 3.1.2. Concretely (for standard safety properties) a program behavior can be understood as an execution trace of the program.

— Sound program analyses *over-approximate* the set of real program behaviors and can prove that a program satisfies a given property (detailed in Section 2.1.1);

— On the contrary, bug-finding techniques, often *under-approximate* the set of real program behaviors. In general, they can not show that a program satisfies a property but they can find real bugs and tend to scale better on common programs (detailed in Section 2.1.2).

In the literature, program analyses have targeted various properties such as memory safety properties (e.g. the absence of use-after-free bugs), functional properties (e.g. partial correctness), timing behavior (e.g. worst-case-execution time analysis), security (e.g. confidentiality of secret data), etc.

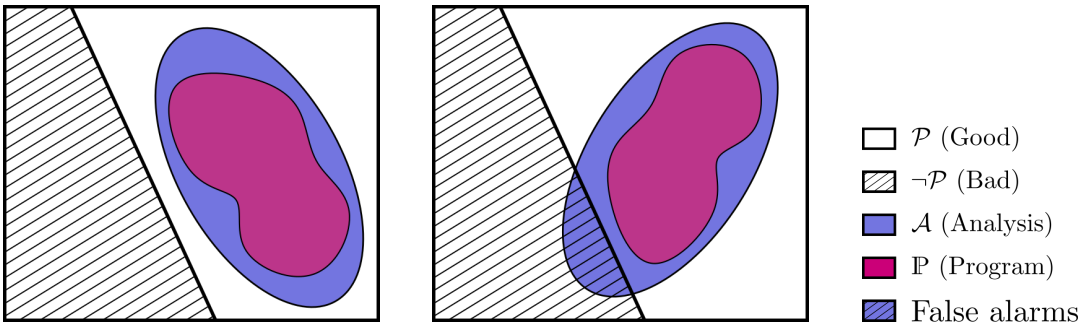
### 2.1.1 Analysis by over-approximation

Sound program analyses provide strong guarantees that a program satisfies a given property by *over-approximating* its semantics, as illustrated in Figure 2.1a. More precisely, the set $\mathcal{A}$ of program behaviors computed by the analysis, encompasses all possible real program behaviors ($\mathbb{P} \subseteq \mathcal{A}$). Consequently, if the analysis can prove that such an over-approximation of the program satisfies a property (i.e. $\mathcal{A} \subseteq \mathcal{P}$) then it can conclude that the program satisfies the property:

$$\mathbb{P} \subseteq \mathcal{A} \wedge \mathcal{A} \subseteq \mathcal{P} \implies \mathbb{P} \subseteq \mathcal{P} \implies \mathbb{P} \vDash \mathcal{P}$$

However, by over-approximating the semantics of a program, the analysis can report false alarms, as illustrated in Figure 2.1b. As a consequence, when such an analysis fails to prove a property, i.e. $\exists c \in \mathcal{A}.\ c \notin \mathcal{P}$, it cannot conclude whether or not the program violates the property. The counterexample could be part of the real program behavior $c \in \mathbb{P}$ or it could be a false alarm $c \notin \mathbb{P}$ and manual analysis is generally required to conclude:

$$\mathbb{P} \subseteq \mathcal{A} \wedge \exists c \in \mathcal{A}.\ c \notin \mathcal{P} \not\implies \exists c \in \mathbb{P}.\ c \notin \mathcal{P}$$




(A) Example of *sound* analysis: the analysis successfully shows that the program is secure.

(B) Example of a *false alarm*: the analysis reports a bug while the program is secure.

FIGURE 2.1 – Illustration of a sound analysis $\mathcal{A}$ for a property $\mathcal{P}$, which *over-approximates* the semantics of a program $\mathbb{P}$.

The main limitation of sound analyses is that when they are applied on programs that have not been designed to be formally verified, the over-approximation can easily become too imprecise, including many behaviors that are not in $\mathbb{P}$. In practice, in this scenario, the analysis will generate false alarms and be unable to prove that the property is satisfied. Moreover, if the program is really insecure, real violations can be

drown in the high number of false alarms making it hard for a developer to conclude if the program indeed violates the property. For these reasons, sound analyses are restricted to (relatively small) critical systems where the violation of a property can have damaging consequences and formal verification is a concern from the start of the development. Successful applications include the ASTRÉE [82] analyzer that has successfully proven the absence of runtime errors in embedded control-command software, or the Frama-C [152] platform which is used for the verification of industrial safety-critical software.

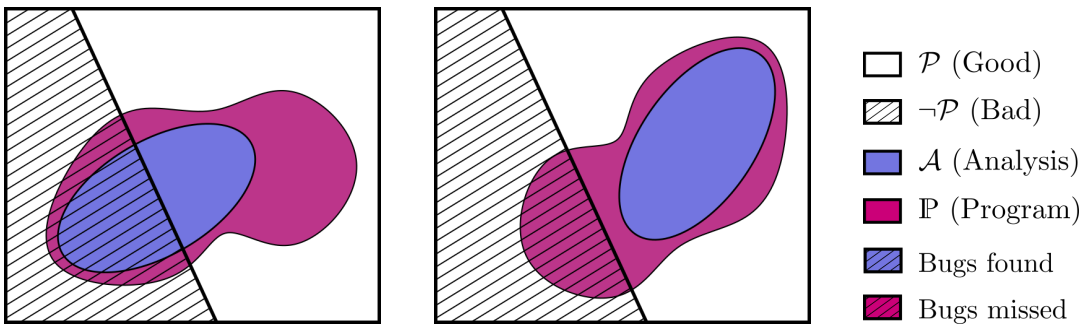
### 2.1.2 Analysis by under-approximation

Contrary to sound analyses, analysis that *under-approximate* the semantics of a program, cannot prove that a program satisfies a property, but can find real violations, as illustrated in Figure 2.2a. More precisely, the set $\mathcal{A}$ of program behaviors computed by the analysis only contains real program behaviors ($\mathcal{A} \subseteq \mathbb{P}$). Consequently, if the analysis finds a bug ($\exists c \in \mathcal{A}.\ c \notin \mathcal{P}$) then it can conclude that the program violates the property:

$$\mathcal{A} \subseteq \mathbb{P} \wedge \exists c \in \mathcal{A}.\ c \notin \mathcal{P} \implies \exists c \in \mathbb{P}.\ c \notin \mathcal{P} \implies \mathbb{P} \not\models \mathcal{P}$$

However, by under-approximating the semantics of a program, the analysis can miss bugs, as illustrated in Figure 2.2b. As a consequence, when such an analysis fails to find bugs, it cannot conclude whether or not the program satisfies the property (the program could satisfy the property or the analysis might have missed a violation):

$$\mathcal{A} \subseteq \mathbb{P} \wedge \mathcal{A} \subseteq \mathcal{P} \not\implies \mathbb{P} \models \mathcal{P}$$




(A) Example of *bug-finding*: the analysis successfully shows that the program is insecure.

(B) Example of *unsound* analysis: the analysis fails to find bugs while the program is insecure.

FIGURE 2.2 – Illustration of bug-finding for a property $\mathcal{P}$ where the analysis $\mathcal{A}$ *under-approximates* the semantics of a program $\mathbb{P}$.

**Definition 3** (Bug-finding)**.** *We use the term* bug-finding *to denote an analysis which is able to conclude that a program violates a property (like under-approximating analyses) but also to report a real program behavior $c \in \mathbb{P} \wedge c \notin \mathcal{P}$ as a counterexample (e.g. a program input whose execution violates the property).*

Contrary to sound analyses, bug-finding techniques are relatively easy to set-up and automate and can often be applied off-the-shelf on large scale systems [121, 48]. They can report concrete bugs to help developers patch their software. These techniques range from random testing to directed fuzzing [119, 57], or coverage guided

fuzzing [58]. For these reason, bug-finding techniques are widely adopted for program analysis, even on non critical code.

**Link with thesis**

In this thesis, we build on an automatic program analysis technique called *symbolic execution*. In general, symbolic-execution *under-approximates* the semantics of programs and is used for bug-finding. However, it can also be used as a sound analysis when it is able to explore all the behaviors of a program.

## 2.2 Symbolic execution

**Section overview**

This section introduces an automated program analysis technique called *symbolic execution* (cf. Section 2.2.1), presents its application for bug-finding and (bounded-)verification (cf. Section 2.2.2), and finally discusses its limitations (cf. Section 2.2.3).

### 2.2.1 Overview of symbolic execution

Symbolic Execution (SE) [151, 63, 59, 121] consists in executing a program on *symbolic inputs* instead of concrete inputs.

Variables and expressions of the program are represented as terms over these symbolic inputs. A *symbolic store*—here denoted $\rho$—maps program variables to their symbolic expressions. Along symbolic execution, when an assignment `x = y + 2` is met, $\rho$ `x` is updated with the symbolic value corresponding to `y + 2`, i.e. $\rho[\texttt{x} \mapsto \rho\ \texttt{y}+2]$.

A *path* in a program is a sequence of instructions that can be executed from the initial state (e.g. the beginning of the main function). It is defined by the control-flow (direction of conditional jumps or targets of indirect jumps). In symbolic execution, the current path is modeled with a logical formula that is the conjunction of conditional expressions (or indirect jump targets) encountered along the execution. This formula is called *path predicate*–here denoted $\pi$. This path predicate can be solved with an off-the-shelf *automated constraint solver*, typically an SMT solver [249], to check if the path is feasible. At a conditional branch, the condition is evaluated to a symbolic expression $c$ and symbolic execution *forks* to follow both outcomes:

— On the first path, the expression $c$ is added to the path predicate (i.e. $\pi = \pi \wedge c$) and the execution moves to the *true* branch;

— On the second path, the negation of the condition $c$ is added to the path predicate (i.e. $\pi = \pi \wedge \neg c$) and the execution moves to the $false$ branch.

Then, symbolic execution continues along satisfiable branches.

Symbolic execution can explore many different program paths, including deep complex paths, and generate concrete test inputs exercising these paths. A illustrated in Example 1, it can also be used to check local assertions along a path, e.g. check at each division `x \ y` that `y` cannot be 0.

> **Example 1** (Symbolic execution)**.** Consider symbolic execution of the simple program in Listing 2.1 where we want to check that the division at line 9 cannot be a division by 0. For simplicity, we assume that integers cannot overflow. Symbolic execution starts with an empty path predicate ($\pi = true$) and a symbolic store that maps input variables `a` and `b` to unconstrained symbolic values

($\rho = \{\mathtt{a} \mapsto a, \mathtt{b} \mapsto b\}$). After the conditional branch at line 2, SE forks into two paths:

1. On the first path, the path predicate is updated to $\pi = a \leq 0$ and the execution continues along the *true* branch. Along this branch, $\rho$ is updated with $\rho[\mathtt{y} \mapsto 1]$ and $\rho[\mathtt{x} \mapsto 0]$. Finally, at line 9, the assertion trivially holds as $\rho\ \mathtt{y} = 1$;
2. On the second path, the path predicate is updated to $\pi = a > 0$ and the execution continues along this branch. At line 5, $\rho$ is updated with $\rho[\mathtt{y} \mapsto 4 \times a]$. At line 6 a new conditional branch is evaluated and forks the execution:
   (a) Along the *then* path, the path predicate is updated to $\pi = a > 0 \wedge b > 0$ and the symbolic store to $\rho[\mathtt{x} \mapsto 2 \times b]$. Finally, at line 9, we need to check that the assertion holds: if it is possible to find input $a$ and $b$ that satisfy $\pi$ and such that $\rho\ \mathtt{y} = 0$ then there is a violation along this path. In other words, we have a violation if $a > 0 \wedge b > 0 \wedge 4 \times a = 0$ is satisfiable—which we can check with a constraint solver. Because the formula is unsatisfiable, there is no violation along this path;
   (b) Along the *else* path, the path predicate is updated to $\pi = a > 0 \wedge b \leq 0$ and the symbolic store to $\rho[\mathtt{x} \mapsto -b]$ and $\rho[\mathtt{y} \mapsto -b \times 4 \times a]$. Finally, at line 9, we check the assertion by sending the query $a > 0 \wedge b \leq 0 \wedge -b \times 4 \times a = 0$ to the solver. The query is satisfiable and the solver returns a counterexample triggering the vulnerability, e.g. $a = 4$ and $b = 0$.

```
1  int foo(int a, int b) {
2    if (a <= 0) {
3      y = 1; x = 0;
4    } else {
5      y = 4 * a;
6      if (b > 0) { x = 2 * b; }
7      else { x = -b; y = x * y; }
8    }
9    return x \ y; // assert y != 0
10 }
```

Listing 2.1 – Example of program.

### 2.2.2 Bug-finding and bounded-verification

Symbolic execution is flexible as it can be used for bug-finding or for sound analysis, offering the guarantees discussed in Sections 2.1.1 and 2.1.2. It can also sacrifice one of these guarantees for instance by unrolling unbounded loops only for a fixed number of iterations, or by using partial concretization/abstractions of the symbolic state [89] (i.e. respectively constraining some variables to be equal to their runtime values / simplifying the formula by replacing complex expressions by unconstrained symbolic values), etc.

**Symbolic execution for bug-finding.** Symbolic execution can be used for bug-finding (unlike most sound program analysis techniques) as it makes no over-approximation when updating the symbolic state. To improve scalability of symbolic

execution, a commonly employed approach is to mix symbolic execution with concrete execution—called dynamic symbolic execution (DSE) [119, 228]. While DSE sacrifices soundness, it greatly improves the robustness of symbolic execution, making it able to scale on larger or more complex programs [48] e.g. by concretely executing system calls that cannot be symbolically executed, or by concretizing part of the formula to help the solver.

**Symbolic execution for (bounded-)verification.** Symbolic execution is sound along a path as it makes no under-approximation when updating the symbolic state. If it can exhaustively explore all program paths, symbolic execution can therefore be used as a sound analysis. In practice, the number and length of paths in a program can be infinite (e.g. because of unbounded loops or recursion) in which case an exhaustive symbolic execution would never terminate. A solution is to consider a weaker form of soundness which restrict to execution traces of size $k$, namely soundness-up-to-$k$—also called *bounded-verification*. Bounded-verification guarantees that a property holds for behaviors of length $k$ (or below). In practice this is enough to prove properties of programs with bounded-loops and recursion such as many cryptographic primitives. To improve the scalability of symbolic execution and allow for (unbounded) sound analyses, it is possible to sacrifice bug-finding, by relaxing the path constraint or by using loop invariants or function summaries [118].

**Symbolic execution in practice.** Dramatic progress in program analysis and constraint solving over the last two decades have made SE a tool of choice for intensive testing. Examples include DSE in SAGE [121], which has proven very successful for testing Microsoft applications in production mode at a very large scale [48]; EXE [60], which has been used to find bugs in Linux file systems [267]; or KLEE [58, 61] which has been successfully applied to generate high coverage tests and to find safety and functional bugs in GNU Coreutils [58, 178], OpenCL programs [79], network protocols implementations [236]. SE is also commonly used in vulnerability analysis [19, 20, 223, 221], firmware analysis [238, 270, 91, 80], and other security-related analysis [264, 27, 263].

**Link with thesis**

Symbolic execution is mostly used for bug-finding and less commonly used for bounded-verification. In this thesis, we are interested in both bug-finding and bounded-verification—which is realistic because the programs we consider are not overly complex (i.e. cryptographic primitives with a few thousand lines). We propose optimizations for symbolic execution that make no over-approximations nor under-approximations along a path.

### 2.2.3 Limitations

Symbolic execution has some limitations that are discussed in detail (together with solutions) in a relatively old but still very relevant survey [63]. Two limitations that are particularly important and that will materialize in this thesis are path explosion and solver limitations.

**Path explosion.** Symbolic execution is vulnerable to *path explosion*, a.k.a. state explosion. As symbolic execution creates a new path at each conditional statement, the number of paths grows exponentially with the program size. If a program contains

unbounded loop or recursion (e.g. `for(i=0;i<n;i++)` when `n` is a symbolic input), the number of paths can even be infinite. To mitigate this problem, a first approach is to identify redundant paths, which will exercise the same behavior as previously explored paths, and prune them [47]. Alternatively, it is also possible to merge different states [161] without giving up on soundness and bug-finding. However, state merging must be applied carefully [160] as it also increases the size and complexity of symbolic expressions.

In addition, search heuristics can be used to guide symbolic execution in order to maximize some coverage criteria [262]. In this case, only a subset of the program paths is explored, which yields to an under-approximation of the program semantic and makes the analysis incomplete, i.e. some bugs can be missed and the program can no longer be proven secure.

**Solver limitations.** Symbolic execution depends on the ability of the solver to solve the symbolic path predicate efficiently. Despite dramatic improvement of solver performance, constraint solving remains the main bottleneck in symbolic execution. As constraints are accumulated over a path, the formula becomes more and more complex, eventually blowing-up the solver on deep and complex paths. To mitigate this issue, a solution is to design simplifications dedicated to specific problems encountered in symbolic execution of real programs [107] or reusing the results of previous similar queries as implemented in KLEE [58].

**Link with thesis**

In Chapter 4, we design dedicated optimizations to resolve constraints ahead of the solver and hence mitigate solver limitations. In Chapter 6, we overcome path explosion due to modeling speculative executions with (sound) redundant path pruning, and a symbolic encoding of memory speculations that avoids path forking.

## 2.3 Binary analysis

**Section overview**

This section, starts by presenting the rationale behind binary analysis (cf. Section 2.3.1), followed by its challenges (cf. Section 2.3.2). Next, it presents the BINSEC binary analysis platform, on top of which we build our analyzers (cf. Section 2.3.3). Finally, it introduces a low-level language called DBA that we use in this thesis to model low-level programs (cf. Section 2.3.4), and presents a symbolic execution for this low-level language (cf. Section 2.3.5).

### 2.3.1 What you see is not what you execute

The semantics of source programs can differ in subtle ways from the semantics of binary code, meaning that analyzing source code can yield different results than analyzing binary code. For instance, scrubbing operations that are used in cryptographic code to clear secret data from memory after the execution of a program can be optimized away by the dead-store-elimination pass of compilers, as detailed in CWE-14 [83]. In the piece of code `memset(key,0,size); free(key)`, the call to `memset` will be removed by the compiler because it has no effect on the program. This is

called the WYSINWYX phenomenon: *What You See (in source code) Is Not What You eXecute* [23].

To address this phenomenon, some tools have taken the path of directly analyzing binary code [96, 53, 230, 171]. These tools disassemble binary code into assembly instructions, lift instructions to an intermediate representation (IR) and perform their analysis at the IR level. They usually come with basic operations on the intermediate language such as control flow graph reconstruction, or code simplifications. Finally, they offer various types of analyses such as symbolic execution, dynamic analysis, tainting, slicing, etc.

Analyzing binary code over source code has many advantages:

— Many programs are distributed in executable form, without access to source code. If users of these programs want to verify the absence of bugs or malicious code (such as backdoors), they have to resort to binary analysis. Binary code analysis is also crucial in malware detection and reverse engineering as their source code is often not available;

— For some vulnerability analysis, source code might simply not be the right level abstraction as many details are only known after compilation. This includes for instance register usage, which is important to reason about secret-erasure as register spilling can store secrets in the memory that are not erased after execution. This also includes the exact sequence of instruction (e.g. whether a piece of code is compiled to branchless code or a conditional jump, or whether an operation is performed via registers or on the stack with `load` and `store` operations) which is important to reason about microarchitectural and transient execution attacks;

— Some properties are not always preserved through compilation, meaning that a property can hold on the source programs but not on the executable code. While there exist certified property-perserving compilers like CompCert [168], they are far from mainstream compilers like `gcc` or `clang` in terms of code optimizations, which strongly restricts their adoption.

**Link with thesis**

In this thesis, we target properties that are not necessarily preserved by compilers. In order to offer strong guarantees on the executable code, we perform our analyses at binary-level. To this end, we build our analyzers on top of the binary analysis platform Binsec [96].

### 2.3.2 Challenges of binary analysis

Low-level code operates on a set of registers and a single (large) untyped memory. During the execution, a call stack contains information about the active functions such as their arguments and local variables. A special register `esp` (stack pointer) indicates the top address of the call stack and local variables of a function can be referenced as offsets from the initial `esp`[2]. Binary analysis poses additional challenges over source-code analysis because many high-level information are lost at compilation, such as types, variables, functions, or control-flow information. Moreover, it has been shown in a recent study [205] that intermediate representations generated from binary code are more verbose and lead to more complex queries than those generated from source code.

2. `esp` is specific to x86, but this is generalizable, e.g. `sp` for ARMv7.

In binary-analysis, control-flow information is not trivial to recover. First, high-level control-flow statement (`while`, `for`, `if`) are translated to *flag* updates, masking the link between conditions and registers or memory locations that set the flag as illustrated in Listing 2.2. This extra level of indirection makes high-level condition recovery challenging [97]. Second, control-flow graph reconstruction relies on the ability of the analysis to identify targets of indirect jumps and function calls [244, 180].

Binary-level analysis requires to explicitly reason about the memory. Evaluation and assignments of source code variables become memory load and store operations in binary code as illustrated in Listing 2.2. Consequently, reconstructing data-dependencies requires to precisely reason on memory accesses. Imprecise reasoning about memory accesses can quickly degenerate and propagate to the whole analysis. Consider for instance a pathological (yet somewhat realistic) case in taint analysis: a `store` operation to an imprecise location can taint the whole memory, making the whole analysis completely impractical as detailed in Example 2.

> **Example 2** (Challenges of binary analysis.)**.** Consider the source code given in Listing 2.2a and its low-level code given in Listing 2.2b.
>
> In the source version, given in Listing 2.2a, we clearly see that `secret` is written in `tab` at line 8 only if the index `idx` is in the bounds of `tab`. Therefore it cannot overwrite the value of the variable `public` at line line 10 and the program is secure.
>
> In the low-level version of the program, given in Listing 2.2b, it is more difficult to conclude. First, the high level condition is not trivial to recover because it involves flag updates via `cmp` instructions. Second, reconstructing the data-flow requires to precisely reason on memory aliases. Consider for instance a taint-based security analysis on this program. To be able to prove the program secure, the analysis must be able to determine a precise address set that is tainted by the store at line 9 and additionally determine that the address of the load at line 10 does not belong to this set. If the analysis is not able to determine a precise address set in which `secret` can be stored at line 9, it would consider that the whole memory is tainted. In this case, the load at line 10 would return a tainted value, triggering a false alarm when passed to the instruction `leak`.

```
1  uint32_t tab[10];
2  uint32_t secret = input();
3  uint32_t public = input();
4  uint32_t idx = input();
5
6  if(0 < idx & idx < 10) {
7    // In-bound store
8    tab[idx] = secret;
9  }
10 leak(public); // secure
11 return 0;
```

(A) Example of secure program.

```
1  cmp   mem[idx], 0 // 0<idx
2  setnz dl
3  cmp   mem[idx], 9 // idx<10
4  setbe al
5  and   eax, edx    // 0<idx<10
6  cmp   eax, 0
7  jz    line 11     // if not(0<idx<10)
8  mov   eax, mem[idx]
9  mov   mem[eax*4 + tab], mem[secret]
10 leak  mem[public]
11 halt
```

(B) Low-level version of Listing 2.2a.

LISTING 2.2 – Example of a program and its compiled version where `secret` and `public` are respectively secret and public input. The program is secure if and only if `secret` cannot leak via the instruction `leak`.

**Link with thesis**

In Chapter 4, we face a comparable challenge where explicitly reasoning about the memory makes the analysis impractical. We propose simplifications dedicated to binary level analysis to mitigate the issue.

Finally, binary analysis tools are also much more difficult and less intuitive to use than their source-level counterparts. For instance, they often require reverse engineering to understand counterexamples or set-up the analysis (e.g. in our case specifying secret data), require special treatments for extra code introduced by compilers (e.g. indirect functions whose implementations are chosen at runtime, stack protectors), are difficult to cross-validate because different tools might not support the same features, etc. We detail some of the challenges we faced when implementing our binary-level analyses, together with the solutions we adopted in Appendix B.

### 2.3.3 BINSEC: a binary-analysis platform

**Section overview**

In this thesis we implement our analyses on top of the binary analysis platform BINSEC [90]. This section introduces the features of BINSEC (cf. Section 2.3.3.1) and its limitations (cf. Section 2.3.3.2).

#### 2.3.3.1 Features

At a high level, BINSEC disassembles binary code to assembly instructions, lifts these instructions to an intermediate language suitable for analysis, called DBA, and provides several analyses operating on DBA. This includes abstract interpretation [96], SE and DSE [90], backward DSE [27], and robust SE [117] [3]. In particular, BINSEC provides the following features, which significantly simplify the development of our analyses:

1. A loader to easily access information encoded in the binary (ELF or PE format) such as segment permissions or symbol information (in non-stripped binaries);
2. Disassembler from x86-32, ARMv7, or RiscV to the DBA intermediate representation [28];
3. A symbolic execution engine—that we adapt for relational symbolic execution in Chapter 4 and exploration of the speculative semantics in Chapter 6;
4. Modules to easily build and manipulate SMT formulas, featuring aggressive simplification [107], and interfacing with many SMT solvers (z3, boolector, cvc4, and yices).

The BINSEC platform has been successfully used to perform various analyses, including bug-finding of use-after-free [108, 188], deobfuscation [27, 220], analysis of inline assembly [211, 210], verification of embedded kernels [189]. Moreover, the lifting to DBA instructions has been positively evaluated in an external study [149].

#### 2.3.3.2 Limitations

**Floating-point operations.** BINSEC does not currently support floating-point instructions. Adding decoding support for decoding floating-point instructions would

3. Robust SE is an adaptation of SE dedicated to robust reachability, a property requiring that a bug is always reachable regardless of uncontrolled inputs such as the initial memory or the value of the initial stack pointer `esp`.

not be difficult. However, supporting floating-point operations in the analyses would be trickier as many constraint solvers still struggle on floating-point reasoning. Until recently, the combination of the floating-point theory with theories used for binary code reasoning (i.e. fixed-size bit-vectors and arrays), namely the QF_ABVFP logic, did not seem to attract much attention (only one benchmark in this division in the 2019 SMT-COMP [234]). However, since 2020, it seems to attract more attention with 500 benchmark in this division in the 2020 SMT-COMP [235] and the apparition of a new solver, `bitwuzla` [190], dedicated to the theories of fixed-size bit-vectors, floating-point arithmetic, and arrays, showing very good performance in this category.

**System calls.** Another limitation of BINSEC is that it does not handle system calls in symbolic execution. A first solution would be to execute system calls concretely in the analysis, however, because it under-approximates the possible program behavior, this solution sacrifices soundness. Another solution would be to provide *stubs* for system call, which model the effect of system calls as transitions on the symbolic state or as sequences of DBA instructions.

**Dynamic memory allocation.** BINSEC symbolic execution engine does not support dynamic memory allocation. A first solution, would be to concretize memory allocations by setting the heap to a concrete address and handling `malloc` calls by allocating disjoint address ranges and returning concrete addresses. The disadvantage of this approach is that it makes assumptions on how the memory is allocated and different allocation choices might lead to different results, making the analysis incomplete. Another solution would be to switch to a region-based memory model, as in CompCert [15], that partitions the memory into distinct regions where each region corresponds to a dynamic allocation. However this makes reasoning about memory accesses and pointers arithmetic more difficult as pointer to different regions cannot be easily compared, requiring dedicated reasoning [96, 42].

**Instruction set architecture.** The BINSEC platform currently only supports x86-32, ARMv7 and RiscV instruction sets. However, because the analysis is performed on the intermediate representation, this limitation is not difficult to overcome. It only requires to implement the decoding of x86-64 and its lifting to DBA expressions [4].

**Dynamic linking.** Finally, BINSEC does not support dynamic libraries so libraries must be statically linked. A first solution would be to implement a dynamic linker, however it is worth noting than function from the standard library are complex (e.g. system calls, indirect functions, etc.) and might pose problem for a static analyzer. A second solution would be to provide stubs for the standard library.

### 2.3.4 Semantics of a low-level language (DBA)

**DBA language.** Through this thesis, we use an intermediate language to model low-level code, called Dynamic Bitvector Automata (DBA) [28], which is used by BINSEC [96] to model low-level programs and perform its analysis. DBA is a general low-level language featuring a small instruction set, which can model common instruction set architecture. DBA instructions are self-contained and free from side-effects unlike real processor instructions, which can for instance implicitly update flags. The

4. As for now, decoding of x86-64 is implemented but testing and experimentation are still in progress.

$$
\begin{array}{rcl}
\mathit{prog} & ::= & \varepsilon \mid \mathit{stmt}\ \mathit{prog} \\
\mathit{stmt} & ::= & \texttt{<l,}\mathit{instr}\texttt{>} \\
\mathit{instr} & ::= & \texttt{v :=}\ \mathit{expr} \mid \texttt{store}\ \mathit{expr}\ \mathit{expr} \mid \texttt{ite}\ \mathit{expr}\ \texttt{?}\ \texttt{l}_1\ \texttt{:}\ \texttt{l}_2 \\
 & \mid & \texttt{goto}\ \mathit{expr} \mid \texttt{goto l} \mid \texttt{halt} \\
\mathit{expr} & ::= & \texttt{v} \mid \texttt{bv} \mid \blacktriangleleft\ \mathit{expr} \mid \mathit{expr}\ \blacklozenge\ \mathit{expr} \mid \texttt{load}\ \mathit{expr} \\
\blacktriangleleft & ::= & \neg \mid - \\
\blacklozenge & ::= & + \mid \times \mid \leq \mid \ldots
\end{array}
$$

FIGURE 2.3 – Syntax of DBA programs, where $\texttt{l}$, $\texttt{l}_1$, $\texttt{l}_2$ are program locations, $\texttt{v}$ is a variable and $\texttt{bv}$ is a value.

lifting of processor instructions to DBA makes all these side-effect explicit, as illustrated in Listing 2.3. These features make the DBA language particularly suitable for formal analysis. The syntax of DBA programs is presented in Figure 2.3.

```
0: res32 := 0x01 + eax;
1: OF := (eax{31} = 0) & (eax{31} != res32{31}); // Overflow flag
2: SF := res32 <s 0; // Sign flag
3: ZF := res32 = 0;  // Zero flag
[...]
7: eax := res32;
```

LISTING 2.3 – Lifting of x86 instruction `add eax, 0x01` to DBA.

**DBA configuration.** Let $\mathit{Inst}$ denote the set of instructions and $\mathit{Loc}$ the set of program locations. A program $\mathbb{P} : \mathit{Loc} \to \mathit{Inst}$ is a map from locations to instructions. Values $\texttt{bv}$ and variables $\texttt{v}$ range over the set of fixed-size bitvectors $BV_n := \{0,1\}^n$ (set of $n$-bit words). A concrete configuration is a tuple $(l, r, m)$ where:

- $l \in \mathit{Loc}$ is the *current location*, and $\mathbb{P}[l]$ returns the current instruction;
- $r : \mathcal{V} \to BV_n$ is a *register map* that maps variables from a set $\mathcal{V}$ to their bitvector value;
- $m : BV_{32} \to BV_8$ is the *memory*, mapping 32-bit addresses to bytes and accessed through instructions `load` and `store`.

An initial configuration is of the form $c_0 \triangleq (l_0, r_0, m_0)$ where $l_0$ is the address of the entrypoint of the program, $r_0$ is an arbitrary register map, and $m_0$ is an arbitrary memory. Let $\mathit{Loc}_\bot \subseteq \mathit{Loc}$ the set of halting program locations such that $l \in \mathit{Loc}_\bot \iff \mathbb{P}[l] = \texttt{halt}$. For the evaluation of indirect jumps, we define a partial one-to-one correspondence from bitvectors to program locations, $\mathit{to\_loc} : BV_{32} \rightharpoonup \mathit{Loc}$. If a bitvector $\texttt{bv}$ corresponds to an illegal location (e.g. non-executable address), $\mathit{to\_loc}\ \texttt{bv}$ is undefined.

**DBA semantics.** The evaluation of a DBA expression $e$ to a bitvector value $\texttt{bv}$ in a configuration $(l, r, m)$, denoted $(l, r, m)\ e \vdash \texttt{bv}$, is detailed in Figure 2.4.

- CST is the evaluation of a constant $\texttt{bv}$ and simply returns the value of the constant;

— VAR is the evaluation of a variable `v` and returns the value mapped to `v` in the register map $r$;

— UNOP is the evaluation of a unary operator $\blacktriangleleft e$. It evaluates the expression $e$ to a concrete value `bv`, and returns the application of the operator $\blacktriangleleft$ to `bv`. The case BINOP, for binary operators, is analogous;

— LOAD is the evaluation of a `load` expression. The rule evaluates the index to a bitvector `bv` and returns the value mapped to `bv` in the memory $m$.

**Expr**

$$\text{CST}\ \frac{}{(l,r,m)\ \texttt{bv} \vdash \texttt{bv}} \qquad \text{VAR}\ \frac{}{(l,r,m)\ \texttt{v} \vdash r\ \texttt{v}}$$

$$\text{UNOP}\ \frac{(l,r,m)\ e \vdash \texttt{bv}}{(l,r,m)\ \blacktriangleleft e \vdash \blacktriangleleft \texttt{bv}} \qquad \text{BINOP}\ \frac{(l,r,m)\ e_1 \vdash \texttt{bv}_1 \qquad (l,r,m)\ e_2 \vdash \texttt{bv}_2}{(l,r,m)\ e_1 \blacklozenge e_2 \vdash \texttt{bv}_1 \blacklozenge \texttt{bv}_2}$$

$$\text{LOAD}\ \frac{(l,r,m)\ e \vdash \texttt{bv}}{(l,r,m)\ \texttt{load}\ e \vdash m\ \texttt{bv}}$$

FIGURE 2.4 – Evaluation of DBA expressions.

The evaluation of the current DBA instruction in a configuration $c$ to a configuration $c'$, denoted $c \to c'$, is detailed in Figure 2.5.

— HALT is the evaluation of a `halt` instruction, and stays on the same final configuration;

— S-JUMP is the evaluation of a static jump. It simply moves control to the next target;

— I-JUMP is the evaluation of an indirect jump. The rule first evaluates the jump target to a bitvector value `bv`, converts it to a location $l'$ and moves control to $l'$. Notice that if `bv` corresponds to an illegal location (i.e. *to_loc* `bv` is undefined), the execution is stuck. For convenience, we restrict to safe programs and, under this hypothesis, *to_loc* `bv` is always defined and the execution is never stuck;

— ITE-TRUE and ITE-FALSE are the evaluation of a conditional jump. If the condition evaluates to *true* (i.e. `bv` $\neq 0$) the rule moves control to $l_1$ whereas if it evaluates to $false$ (i.e. `bv` $= 0$) the rule moves control to $l_2$;

— ASSIGN is the evaluation of an assignment to a variable `v`. It evaluates the left-hand side expression to a bitvector value `bv` and updates the value of `v` in the register map $r$ to map to `bv`. Finally, it moves control to the next instruction;

— STORE is the evaluation of a store instruction. The rule first evaluates the index $e$ and the value to store $e'$ to bitvector values `bv` and `bv'`. Then, it updates the index `bv` in the memory $m$ to map to `bv'`. Finally, it moves control to the next instruction.

### 2.3.5 Binary-level symbolic execution

Symbolic execution scales relatively well on binary code—due to both the efficiency of SMT solvers and concretization. Hence, strong binary-level SE tools do exist and have yielded several highly promising case studies [121, 19, 72, 230, 90, 27, 220].

**Instr**

$$\text{HALT}\quad \frac{\mathbb{P}[l] = \texttt{halt}}{(l, r, m) \to (l, r, m)} \qquad \text{S-JUMP}\quad \frac{\mathbb{P}[l] = \texttt{goto}\ l'}{(l, r, m) \to (l', r, m)}$$

$$\text{I-JUMP}\quad \frac{\mathbb{P}[l] = \texttt{goto}\ e \qquad (l, r, m)\ e \vdash \texttt{bv} \qquad l' \triangleq to\_loc(\texttt{bv})}{(l, r, m) \to (l', r, m)}$$

$$\text{ITE-TRUE}\quad \frac{\mathbb{P}[l] = \texttt{ite}\ e\ \texttt{?}\ l_1\ \texttt{:}\ l_2 \qquad (l, r, m)\ e \vdash \texttt{bv} \qquad \texttt{bv} \neq 0}{(l, r, m) \to (l_1, r, m)}$$

$$\text{ITE-FALSE}\quad \frac{\mathbb{P}[l] = \texttt{ite}\ e\ \texttt{?}\ l_1\ \texttt{:}\ l_2 \qquad (l, r, m)\ e \vdash \texttt{bv} \qquad \texttt{bv} = 0}{(l, r, m) \to (l_2, r, m)}$$

$$\text{ASSIGN}\quad \frac{\mathbb{P}[l] = \texttt{v := } e \qquad (l, r, m)\ e \vdash \texttt{bv}}{(l, r, m) \to (l + 1, r[\texttt{v} \mapsto \texttt{bv}], m)}$$

$$\text{STORE}\quad \frac{\mathbb{P}[l] = \texttt{store}\ e\ e' \qquad (l, r, m)\ e \vdash \texttt{bv} \qquad (l, r, m)\ e' \vdash \texttt{bv}'}{(l, r, m) \to (l + 1, r, m[\texttt{bv} \mapsto \texttt{bv'}])}$$

FIGURE 2.5 – Evaluation of DBA instructions.

**Logical notations.** Low-level code operates on a set of registers and a single large memory. In binary-level symbolic execution, values (e.g. registers, memory addresses, memory content) are modeled with fixed-size bitvectors [111]. We use the type $\mathcal{B}v_m$, where $m$ is a constant number, to represent symbolic bitvector expressions of size $m$. The memory is modeled with a logical array [16] of type $(Array\ \mathcal{B}v_{32}\ \mathcal{B}v_8)$ (assuming a 32-bit architecture).

The logic that combines reasoning on fixed-size bitvectors and array is called QF_ABV [29] (quantifier-free formulas over the theory of fixed-size bitvectors and array). We let $\Phi$ denote the set of symbolic expressions in the QF_ABV logic and $\varphi$, $\phi$, $\psi$, $\iota$ be symbolic expressions ranging over $\Phi$.

A symbolic array is a function $(Array\ \mathcal{I}\ \mathcal{V})$ that maps each index $i \in \mathcal{I}$ to a value $v \in \mathcal{V}$. Operations over arrays are:

— $select : (Array\ \mathcal{I}\ \mathcal{V}) \times \mathcal{I} \to \mathcal{V}$: take as arguments an array $a$ and an index $i$ and returns the value $v$ stored at index $i$ in $a$;

— $store : (Array\ \mathcal{I}\ \mathcal{V}) \times \mathcal{I} \times \mathcal{V} \to (Array\ \mathcal{I}\ \mathcal{V})$: take as arguments an array $a$, an index $i$, and a value $v$, and returns the array $a$ modified so that $i$ maps to $v$.

These functions satisfy the following constraints for all array $a \in (Array\ \mathcal{I}\ \mathcal{V})$, indexes $i, j \in \mathcal{I}$, and value $v \in \mathcal{V}$:

— $select\ (store\ a\ i\ v)\ i = v$: a *store* of a value $v$ at index $i$ followed by a *select* at the same index $i$ returns the value $v$;

— $i \neq j \implies select\ (store\ a\ i\ v)\ j = select\ a\ j$: a *store* at an index $i$ does not affect values stored at other indexes $j$.

**Satisfiability of a formula.** A formula $\pi$ is *satisfiable* if there exists a model $M$ that assigns concrete values to symbolic variable such that the formula $\pi$ is *true*; if there is no such assignment, the formula is *unsatisfiable*. The satisfiability of a formula $\pi$ with a model $M$ is denoted $M \vDash \pi$. By extension, we let $M(\varphi)$ denote the concrete evaluation of a symbolic expression $\varphi$ (whose free variables are defined in $pi$) under the assignment $M$. In the implementation, an SMT solver is used to determine the satisfiability of a formula and obtain a satisfying model, denoted $M \vDash_{\text{SMT}} \pi$. Whenever the model is not needed for our purposes, we leave it implicit and simply write $\vDash \pi$ or $\vDash_{\text{SMT}} \pi$ for satisfiability.

**Symbolic configuration.** A *symbolic configuration* is of the form $(l, \rho, \mu, \pi)$ where:

— $l \in Loc$ is the current program point;

— $\rho : \mathcal{V} \to \Phi$ is a symbolic register map, mapping variables from a set $\mathcal{V}$ to their symbolic representation as a symbolic expression in $\Phi$;

— $\mu : (Array\ \mathcal{B}v_{32}\ \mathcal{B}v_8)$ is the symbolic memory—an array of values in $\mathcal{B}v_8$ indexed by addresses in $\mathcal{B}v_{32}$;

— $\pi \in \Phi$ is the path predicate—a conjunction of conditional statements and assignments encountered along a path.

The location $l$ is not needed for symbolic evaluation of expressions, therefore, we omit it and write $(\rho, \mu, \pi)$ to denote a symbolic configuration in this context.

**Symbolic evaluation.** Symbolic evaluation of an expression *expr* in a configuration $(\rho, \mu, \pi)$ to a formula $\varphi$, is denoted $(\rho, \mu, \pi)\ expr \vdash \varphi$ and is given in Figure 2.6. Detailed explanations of the rules follow:

— CST is the evaluation of a constant `bv` and returns the corresponding symbolic bitvector *bv*;

— VAR is the evaluation of a variable `v` and returns the value mapped to `v` in the register map $\rho$;

— UNOP is the evaluation of a unary operator $\blacktriangleleft\ e$. It evaluates the expression $e$ to a symbolic value $\varphi$, and returns the application of the symbolic operator $\triangleleft$ (corresponding to the concrete operator $\blacktriangleleft$) to $\varphi$. The case BINOP, for binary operators, is analogous;

— LOAD is the evaluation of a load expression. The rule computes the symbolic index $\iota$ and returns a logical *select* expression from the symbolic memory $\mu$ at index $\iota$.

Symbolic evaluation of instructions, denoted $s \rightsquigarrow s'$ where $s$ and $s'$ are symbolic configurations, is given in Figure 2.7. Detailed explanations of rules follow:

— S-JUMP is the evaluation of a static jump. It simply moves control to the next target;

— I-JUMP is the evaluation of an indirect jump. The rule first evaluates the jump target to a symbolic expression $\varphi$. It finds a concrete value $l'$ for the jump target that satisfies the path predicate, and updates the path predicate and the next location accordingly. Note that this rule is nondeterministic as $l'$ can be any concrete value satisfying the constraint. In practice, we call the solver to enumerate jump targets up to a given bound and continue the execution along valid targets (which jump to an executable section);

**Expr**

$$\textsc{cst}\ \frac{}{(\rho,\mu,\pi)\ \texttt{bv} \vdash bv} \qquad \textsc{var}\ \frac{}{(\rho,\mu,\pi)\ \texttt{v} \vdash \rho\ \texttt{v}}$$

$$\textsc{unop}\ \frac{(\rho,\mu,\pi)\ e \vdash \phi \qquad \varphi \triangleq \triangleleft\, \phi}{(\rho,\mu,\pi)\ \blacktriangleleft\, e \vdash \varphi}$$

$$\textsc{binop}\ \frac{(\rho,\mu,\pi)\ e_1 \vdash \phi \qquad (\rho,\mu,\pi)\ e_2 \vdash \psi \qquad \varphi \triangleq \phi \diamond \psi}{(\rho,\mu,\pi)\ e_1 \blacklozenge e_2 \vdash \varphi}$$

$$\textsc{load}\ \frac{(\rho,\mu,\pi)\ e_{idx} \vdash \hat{\imath} \qquad \varphi \triangleq select(\mu,\iota)}{(\rho,\mu,\pi)\ \texttt{load}\ e_{idx} \vdash \varphi}$$

FIGURE 2.6 – Symbolic evaluation of DBA expressions where $\triangleleft$ (resp. $\diamond$) is the logical counterpart of the concrete operator $\blacktriangleleft$ (resp. $\blacklozenge$).

— ITE-TRUE is the evaluation of a conditional jump when the expression evaluates to *true* (the $false$ case is analogous). The rule first evaluates the condition to a symbolic expression $\varphi$, and if condition guarding the *true*-branch is satisfiable (i.e. $\vDash_{\text{SMT}} \pi \wedge (\varphi \neq 0)$), the rule updates next location to explore it;

— ASSIGN is the evaluation of an assignment. It allocates a fresh symbolic variable to avoid term-size explosion, and updates the register map and the path predicate;

— STORE is the evaluation of a store instruction. The rule evaluates the index and value of the store, and updates the symbolic memory and the path predicate with a logical *store* operation.

**Link with thesis**

In this thesis, we extend binary-level symbolic execution presented in this section in two directions. In Chapters 4 and 5, we propose extensions to model pairs of traces efficiently and verify information-flow policies. In Chapter 6, we propose extensions to account for the speculative semantics and detect Spectre attacks.

**Instr**

$$\text{S\_JUMP}\ \frac{\mathbb{P}[l] = \texttt{goto}\ l'}{\big(l, \rho, \mu, \pi\big) \rightsquigarrow \big(l', \rho, \mu, \pi\big)}$$

$$\text{I\_JUMP}\ \frac{\mathbb{P}[l] = \texttt{goto}\ e \qquad \big(\rho, \mu, \pi\big)\ e \vdash \varphi \qquad M \vDash_{\text{SMT}} \varphi \qquad l' \triangleq to\_loc(M(\varphi)) \qquad \pi' \triangleq \pi \wedge (\varphi = M(\varphi))}{\big(l, \rho, \mu, \pi\big) \rightsquigarrow \big(l', \rho, \mu, \pi'\big)}$$

ITE-TRUE

$$\frac{\mathbb{P}[l] = \texttt{ite}\ e\ \texttt{?}\ l_{true}\ \texttt{:}\ l_{false} \qquad \big(\rho, \mu, \pi\big)\ e \vdash \varphi \qquad \pi' \triangleq \pi \wedge (\varphi \neq 0) \qquad \vDash_{\text{SMT}} \pi'}{\big(l, \rho, \mu, \pi\big) \rightsquigarrow \big(l_{true}, \rho, \mu, \pi'\big)}$$

ITE-FALSE

$$\frac{\mathbb{P}[l] = \texttt{ite}\ e\ \texttt{?}\ l_{true}\ \texttt{:}\ l_{false} \qquad \big(\rho, \mu, \pi\big)\ e \vdash \varphi \qquad \pi' \triangleq \pi \wedge (\varphi = 0) \qquad \vDash_{\text{SMT}} \pi'}{\big(l, \rho, \mu, \pi\big) \rightsquigarrow \big(l_{false}, \rho, \mu, \pi'\big)}$$

$$\text{ASSIGN}\ \frac{\mathbb{P}[l] = \texttt{v}\ \texttt{:=}\ e \qquad \big(\rho, \mu, \pi\big)\ e \vdash \varphi \qquad \varphi' \triangleq \mathit{fresh} \qquad \rho' \triangleq \rho[\texttt{v} \mapsto \varphi'] \qquad \pi' \triangleq \pi \wedge (\varphi' = \varphi)}{\big(l, \rho, \mu, \pi\big) \rightsquigarrow \big(l + 1, \rho', \mu, \pi'\big)}$$

STORE

$$\frac{\mathbb{P}[l] = \texttt{store}\ e_{idx}\ e_{val} \qquad \big(\rho, \mu, \pi\big)\ e_{idx} \vdash \iota \qquad \big(\rho, \mu, \pi\big)\ e_{val} \vdash \nu \qquad \mu' \triangleq store(\mu, \iota, \nu) \qquad \pi' \triangleq \pi \wedge \mu' = store(\mu, \iota, \nu)}{\big(l, \rho, \mu, \pi\big) \rightsquigarrow \big(l + 1, \rho, \mu', \pi'\big)}$$

FIGURE 2.7 – Symbolic evaluation of DBA instructions and expressions where *fresh* returns a new unconstrained symbolic variable.

# Chapter 3

# Low-level Security

**Chapter overview**

This chapter introduces technical background on the properties that we analyze in this thesis.

— Section 3.1 introduces information flow properties, highlight the fact that these properties are not regular safety but 2-hypersafety properties—i.e. relating pairs of traces—and gives an overview of existing verification techniques;

— Section 3.2 defines timing attacks and gives an overview of microarchitectural timing attacks. It also introduces the *constant-time* property, a software countermeasure that protects against microarchitectural timing attacks and which we analyze in this thesis;

— Section 3.3 defines transient execution attacks and presents details of the microarchitecture that are necessary for the comprehension of these attacks. It also introduces the *speculative constant-time* property, the extension of constant-time to transient execution attacks which we analyze in this thesis.

## 3.1 Information flow properties

**Section overview**

This section first introduces information flow policies and defines a well known information flow property called *noninterference*, which is the basis of the properties considered in this thesis (cf. Section 3.1.1). Second, it highlights the fact that these properties are not regular safety but 2-hypersafety properties—i.e. relating pairs of traces (cf. Section 3.1.2). It finally gives an overview of existing verification techniques for information flow policies (cf. Section 3.1.3).

### 3.1.1 Definition of information flow and noninterference

*Information flow* policies regulate the transfer of information between public and secret domains. These policies are crucial to express important security policies such as confidentiality—an attacker cannot infer secret input data by looking at the observable outputs of the system. To reason about information flow, the program input is usually

partitioned into two disjoint sets: the set of *low* (or public) input, and the set of *high* (or secret) input [1].

Intuitively, information may flow from public to secret domain, as illustrated in Listing 3.1a; but information flow from high to low domain is forbidden, as illustrated in Listing 3.1b. Information may also flow indirectly via the control flow of the program, as illustrated in Listing 3.1c where the final value of the public variable `l` reveals the value of `h mod 2`. Finally, information can also leak via other mechanisms that are not primarily intended for communicating information, called *side channels*. This includes for instance program termination, as illustrated in Listing 3.1d where the termination of the program leaks the value of `h mod 2`—i.e. the program loops forever if `h mod 2 = 1` and terminates otherwise. *Timing channel* can also leak information, as illustrated in Listing 3.1e where the execution time of the program depends on the value of `h mod 2`.

```
h = l mod 2
```

(A) Legal information flow.

```
l := h mod 2
```

(B) Leak via direct information flow.

```
h := h mod 2;
if (h = 1) then l := 1
else l := 0
```

(C) Leak via indirect information flow.

```
h := h mod 2;
while (h = 1) do skip
```

(D) Leak via termination channel.

```
h := h mod 2;
if h = 1 then sleep 5
else skip
```

(E) Leak via timing channel.

LISTING 3.1 – Examples of programs to illustrate information flow control where `h` is a high variable and `l` is a low variable.

**Noninterference.** *Noninterference* [122] is a well known information flow property requiring that the secret input of the program does not interfere with publicly observable outputs.

**Definition 4** (Noninterference)**.** *A program is noninterferent if for each pair of initial configurations $c$ and $c'$ that agree on their low input (denoted $c \simeq_l c'$) and that evaluate to final configurations $c_f$ and $c'_f$ (denoted $c \rightarrow^* c_f$), then $c_f$ and $c'_f$ must also agree on their low output (denoted $c_f \simeq_o c'_f$). Formally:*

$$c \simeq_l c' \wedge c \rightarrow^* c_f \wedge c' \rightarrow^* c'_f \implies c_f \simeq_o c'_f$$

**Example 3** (Noninterference)**.** Consider the examples given in Listings 3.1a to 3.1c where both equality of public inputs $\simeq_l$ and public outputs $\simeq_o$ are defined as the equality of the low variable `l`.

The program in Listing 3.1a is trivially noninterferent because low variable `l` is never assigned. Therefore, for two executions of the program starting with the same value for `l`, the values of `l` in the final configurations are also the same,

1. In this thesis, we restrict to this simple high-low policy but note that it can be generalized to a lattice of security domains [37, 93] where ordering relations between security domains in the lattice determine legal flows.

regardless of the value of `h`.

On the contrary, programs in Listings 3.1b and 3.1c are not noninterferent because the final value of `l` depends on the initial value of `h`. Take for instance, two initial configurations:

— $c$ where `h = 0` and `l = 0`, which produces a final configurations $c_f$ where `l = 0`;

— $c'$ where `h = 1` and `l = 0`, which produces a final configuration $c'_f$ where `l = 1`.

We have exhibited two executions that agree on their initial low input but produce different low outputs which violate noninterference.

**Link with thesis**

In this thesis, we design symbolic analyzers for properties derived from noninterference.

### 3.1.2 Noninterference is a 2-hypersafety property

Contrary to standard *safety* properties which state that nothing bad can happen along *one execution* of a program, information flow properties relate *two execution traces*—they are *2-hypersafety properties* [77]. In this section we give a high-level definition of *safety* and *2-hypersafety*, a reader interested by the formal definition of these notions can read the excellent paper from Clarkson and Schneider [78] that introduces hyperproperties.

**Safety properties.** Safety properties state that "bad things" cannot happen during the execution of a program. They have first been introduced by Lamport [165] together with *liveness* properties, which state that "something good" will eventually happen [2]. According to Alpern and Schneider [12], a "bad thing" must be:

— finitely observable: it must occur within a finite trace prefix,

— irremediable: once an execution has violated the property, any extension of this execution still violates the property.

A counterexample of a safety property is therefore a finite trace prefix in which a "bad thing" occurs. For instance, the absence of runtime errors is a safety property and a counterexample is an execution of a program that triggers a runtime error.

Safety (and liveness) properties are *trace properties*, i.e. properties of individual execution traces. However, noninterference is a property that cannot be expressed as a trace property as it relates two executions of a program: it is a *2-hypersafety property*.

**2-hypersafety properties.** Hyperproperties [78] generalize trace properties from properties of individual traces to properties of *sets of traces*. In this thesis we focus on a subset of hyperproperties, namely *2-hypersafety properties* which generalize safety properties to *pairs of traces* and are sufficient to express the information flow properties we consider.

In *2-hypersafety properties*, the "bad thing" is generalized from a finite trace to a *pair of finite traces*, meaning that a counterexample of an hypersafety property is a

2. In this thesis, we focus on safety properties but the reader interested in liveness properties can in Alpern and Schneider's work, the first formal definition of liveness [11] and how to prove safety and liveness [12]

pair of execution traces. For instance, noninterference as defined in Definition 4 is a 2-hypersafety property and a counterexample is a *pair of executions* that initially agree on their low input but end with distinct low outputs.

### 3.1.3 Verification of information flow properties

Information flow type systems have been a widely studied approach for enforcing information flow [218]. However, while these type systems can prove that a program is secure, they are also prone to false alarms.

Alternatively, Barthe, D'Argenio, and Rezk showed that it is possible to reduce verification of a 2-hypersafety property to verification of a safety property of a transformed program via *self-composition* [33].

**Lifting 2-hypersafety to safety via self-composition.** Because information-flow properties are not standard safety properties but 2-hypersafety properties[3], verification tools designed for safety do not directly apply to 2-hypersafety. *Self-composition* reduces a 2-hypersafety property of a program $\mathbb{P}$ to a safety property of a transformed program $\mathbb{P};\mathbb{P}'$ where $\mathbb{P};\mathbb{P}'$ is the composition of $\mathbb{P}$ with a renamed version of itself, as illustrated in Example 4.

**Example 4** (Self-composition)**.** Let the original program $\mathbb{P}$ be the program in Listing 3.2a. The self-composed version of $\mathbb{P}$ is the program $\mathbb{P};\mathbb{P}'$ given in Listing 3.2b.

Noninterference of $\mathbb{P}$ can be reduced to the following safety property of program $\mathbb{P};\mathbb{P}'$:

**Definition 5** (Noninterference of $\mathbb{P}$ by self-composition)**.** *Program* $\mathbb{P}$ *is noninterferent if and only if for all initial configuration* $c_0$ *of* $\mathbb{P};\mathbb{P}'$*, if* `l = l'` *and* $c_0$ *executes to a final configuration* $c_f$*, then* `l = l'` *in* $c_f$*.*

It is easy to show that the property does not hold by taking an initial configuration with `l = l' = 1`, `h = 1`, and `h' = 0`. This produces a final configuration with `l = 1` and `l' = 0`, which violates Definition 5.

```
h := h mod 2;
if (h = 1) then l := 1
else l := 0
```

(A) Example of program.

```
h := h mod 2;
if (h = 1) then l := 1
else l := 0;
h' := h' mod 2;
if (h' = 1) then l' := 1
else l' := 0
```

(B) Self composed version of program in Listing 3.2a.

LISTING 3.2 – Example of (sequential) self-composition where `h` is a high variable and `l` is a low variable.

3. In this thesis we consider only information flow policies expressible as 2-hypersafety. There exist information flow policies that are not 2-hypersafety, such as possibilistic noninterference. However, for simplicity, when we mention information-flow properties, we refer to those expressible as 2-hypersafety.

By reducing verification of 2-hypersafety to verification of safety, self-composition makes it possible to reuse off-the-shelf tools designed for safety analysis in the context of 2-hypersafety. Unfortunately, it has been showed that this approach does not scale because off-the-shelf tools do not exploit redundancies inherent to self-composed programs [241]. To reduce the complexity of verifying the self-composed programs and improve the applicability of self-composition, many approaches have been proposed such as type-directed self-composition [241], product programs [32], modular product programs [102], lazy-self composition [268], property-directed self-composition [229], etc. Another approach is to directly adapt existing techniques to support reasoning about two executions, such as relational Hoare logic [38], probabilistic relational Hoare logic [34], cartesian Hoare logic [237], relational separation logic [266], relational symbolic execution [105], multiple facets [18, 187].

**Link with thesis**

In this thesis, we build on relational symbolic execution, a recent adaptation of symbolic execution to information flow analysis.

## 3.2 Timing and microarchitectural attacks

Timing attacks and microarchitectural attacks belong to the class of side-channel attacks. *Side-channel attacks* exploit the physical parameters of the implementation or a cryptosystem rather than its mathematical properties. *Timing attacks* exploit secret-dependent variations of the execution time of a program. *Microarchitectural timing attacks* are a special case of timing attacks that exploit secret-dependent changes to the microarchitectural state (such as the cache) that can be exploited through timing to recover secret data. Unlike other side-channel attacks (e.g. power consumption [157] or electromagnetic emissions [209]), timing attacks do not require special equipment or physical access to the machine [52] and are therefore particularly devastating.

**Section overview**

This section first defines timing attacks (cf. Section 3.2.1), then details *cache attacks*, the most common type of microarchitectural attack (cf. Section 3.2.2), and gives an overview of other microarchitectural timing attacks (cf. Section 3.2.3). Finally, it defines *constant-time* a software-based countermeasure against timing attacks that we analyze in this thesis (cf. Section 3.2.4).

### 3.2.1 Timing attacks

Execution time of a program can vary from one input to another according to multiple parameters: control-flow, cache hit and misses, instructions that execute in variable time depending on the value of operands (such as division), processor optimizations, etc. Timing attacks exploit secret-dependent variations of execution time in order to infer secret data using statistical analysis. Concretely, when the execution time of a program depends on the value of a secret input, an attacker can exploit the timing variation to recover the value of a secret, as coarsely illustrated in Figure 3.1.

Timing attacks have been theorized by Paul Kocher in 1996 [156]. He designed a timing attack targeting the modular exponentiation algorithm, which is at the root

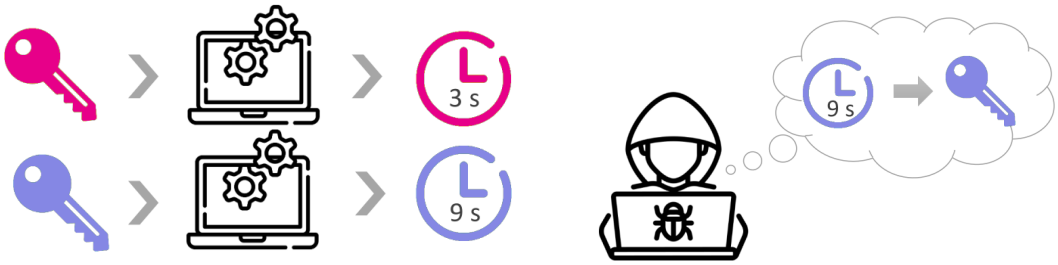



FIGURE 3.1 – Illustration of secret-dependent execution time. Clock made by bqlqn and other icons made by Freepik from www.flaticon.com.

of RSA and Diffie-Hellman popular cryptosystems. Shortly after, the attack has been practically demonstrated, enabling full recovery of RSA secret keys in smartcards [94].

In the RSA cryptosystem, the decryption of a ciphertext $c$ with a public key $k$ is given by the modular exponentiation $M = c^k \bmod N$. The ciphertext $c$ can be chosen by an attacker; $N$ is a public exponent, known by the attacker; and $k$ is the secret key that the attacker wants to recover. Fast modular exponentiation can be implemented using the square and multiply algorithm given in Algorithm 1. Notice that at step $i$, an extra step is performed depending on the value of the secret bit $k_i$. This extra step introduces a secret-dependent timing variation that can be exploited by an attacker to recover the secret key one bit at a time.

**Input:** Ciphertext $c$, Secret key $k$.
**Result:** $c^k \bmod N$
$x \leftarrow c$;
**for** $i = 1$ **to** $w$ **do**
  $x \leftarrow x^2 \bmod N$;
  **if** $k_i == 1$ **then** ▷ *Extra modular multiplication if $i^{th}$ bit of key is 1*
    $x \leftarrow x \times c \bmod N$;
**return** $x$

**Algorithm 1:** Square and multiply algorithm where $w$ is the size (in bits) of the key $k$ and $k_i$ is the i$^{\text{th}}$ bit of $k$.

For instance, an attacker can infer the value of the second bit [4], $k_2$, as follows:

— If $k_2 = 1$ then the algorithm performs a square-and-multiply operation, i.e. $x \leftarrow c^2 \bmod N$ followed by $x \leftarrow x \times c \bmod N$;

— If $k_2 = 0$ then the algorithm only performs a square operation $x \leftarrow c^2 \bmod N$.

Let us consider that the attacker can craft two ciphertexts $c_{fast}$ and $c_{slow}$ such that computing the extra multiplication $x \leftarrow x \times c_{fast} \bmod N$ is fast and $x \leftarrow x \times c_{slow} \bmod N$ is slow [5]. Using statistical analysis, the attacker can infer the value of the secret bit $k_2$ from timing variations between $c_{fast}$ or $c_{slow}$:

— If the multiplication is performed (i.e. $k_2 = 1$), the computation will be faster for $c_{fast}$ than for $c_{slow}$;

— If the multiplication is not performed (i.e. $k_2 = 0$), the timing variation when computing $c_{fast}$ and $c_{slow}$ should look random.

4. Supposing that the first bit of $k$ is always 1.

5. This can be done by choosing $c_{fast}$ and $c_{slow}$ such that the result of the multiplication $x \times c_{slow}$ is greater than the exponent $N$—thus requiring an extra reduction step—and $x \times c_{fast}$ is lower than $N$—thus sparing this extra reduction step.

Once bits $k_0 \dots k_i$ have been recovered, the attacker can use the same method to retrieve bit $k_{i+1}$, and eventually reconstruct the secret key one bit at a time.

*In this particular case, a timing channel can be exploited because the sequence of executed instructions depends on the secret.*

### 3.2.2 Cache attacks

CPU caches are used to reduce the latency of memory accesses and mitigate the performance gap between CPU and accesses to the main memory. Cache are small fast memories located close to processor cores, which store data that have been recently accessed from the main memory. When a memory access is performed, the processor first checks if the corresponding entry is already in the cache. If the entry is in the cache, its value is directly returned from the cache, this is called a *cache hit*. Otherwise, the data has to be read from the (slower) main memory, it is called a *cache miss*. After a cache miss, a new entry is inserted into the cache to hold the data, evicting another entry [6]. *Cache attacks* exploit these timing variations, induced by cache hits and misses, in order to infer information on the memory accesses of a victim and recover secret data.

**Cache hierarchy.** Most CPU have three levels of caches L1, L2, L3 where the L1 cache is the fastest but also the smallest, whereas the L3 cache is the biggest but also the slowest (but still much faster than the main memory). L1 and L2 caches are usually private to a CPU core, whereas the L3 cache is shared between cores, enabling cross-cores attacks.

**Cache attacks** There are many types of cache attacks with different requirements and targeting different levels of the cache hierarchy. We only detail here two well known attacks (Prime+Probe and Flush+Reload) to illustrate different mechanisms by which an attacker can recover secret via the cache side-channel. The interested reader can find a more complete overview of existing attacks (and defenses) in van Schaik et al.'s work [248].

**Prime and probe [195].** Prime+Probe has no strict requirement as the attacker only needs to share a cache with its victim. Originally, Prime+Probe attack targeted the L1 cache [195] (thus additionally requiring the attacker to execute on the same physical core as the victim) but further versions demonstrate that it can also be used against L3 cache, enabling cross-cores and cross-VM attacks [174].

1. The attacker *primes* the cache by filling it with her own entries;
2. The attacker executes the victim's code;
3. Once the victim has executed, the attacker can *probe* the cache by timing accesses to her previously loaded lines. If the attacker observes a cache miss, then her entry has been evicted, meaning that it has been accessed by the victim; whereas if she observes a cache hit, the line has not been accessed by the victim.

**Flush and reload [131].** Flush+Reload has stricter requirements than Prime+Probe as it additionally requires shared memory between the attacker and the victim [7] and the ability to flush instruction—e.g. using the `cflush` instruction.

6. Cache placement policies (how addresses are mapped to cache lines) and replacement policies (which cache lines are evicted in case of conflict) are not detailed here.

7. Identical pages, e.g. shared libraries, can be shared between processes to reduce the memory footprint.

1. The attacker *flushes* some cache lines that she shares with the victim;
2. The attacker executes the victim's code;
3. Once the victim has executed, the attacker can measure the time taken to *reload* the cache lines that she has flushed. If she observes a cache hit, she can deduce that the victim has accessed the shared line; whereas if she observes a cache miss, she can deduce that the shared line has not been accessed by the victim.

**Leaking secret via cache attacks.** Using cache attacks, like PRIME+PROBE and FLUSH+RELOAD, an attacker can deduce which cache lines have been accesses by her victim and reconstruct the memory addresses accessed by her victim. If the victim's memory accesses depend on secret data, an attacker can thus recover these secret data. For instance, cache attacks have been successfully mounted to expose AES secret keys [131, 195, 39, 177], break ASLR [124], or even track users' behavior from malicious JavaScript [194].

### 3.2.3 Other microarchitectural side-channels

As research on microarchitectural side-channel evolves, new variants of microarchitectural channels are discovered. Apart from CPU caches, programs can leave observable traces in many components of the microarchitecture, forming a wide variety of microarchitectural side-channel that we cannot cover exhaustively. We still give an overview of some of these attacks to illustrate their diversity and we refer the interested reader to a survey [115] for more details on microarchitectural timing attacks (and defenses).

The microarchitectural state includes many caches that, when shared between an attacker and her victim, can be exploited to leak information. This includes for instance DRAM row buffers [200] (which store the last accessed row in a DRAM bank), instruction caches [2] (which store recently executed instructions), translation lookaside buffers (TLB) [123] (which store translations between virtual addresses and physical addresses), micro-op caches [213] (which store the translation of instructions into micro-ops), cache directories [265] (which are used in cache coherence protocols to track resident lines in the cache hierarchy) etc. In addition to timing attacks, the cache state can also be extracted via alias-driven attacks, which exploit incoherent memory states (e.g. resulting from aliases between cacheable and non-cacheable virtual addresses) to observe which addresses are accessed by a victim [128].

In modern processors, the microarchitecture also includes many predictors that improve performance but can leak information on the data they are trained with. Such side-channels attacks have been demonstrated on the direct branch predictor [3, 104], indirect branch predictor [103], or return predictor [56]. Attackers can exploit port contention to the execution units in hyperthreaded environments [5], revealing which execution units are used by the victim (and therefore part of the control flow). Non-constant time operations can leak content about their operands via timing such as early-terminating multiplications [125] in embedded processors or floating-point operations [14]. Finally, some microarchitectural side-channel attacks exploit the behavior of hardware extensions. This includes for instance an attack abusing abort operations in Intel Transactional Synchronization Extension (Intel TSX), which de-randomizes kernel memory layout and breaks KASLR [145]; or detecting whether the advanced extension (AVX) unit is active in order to leak parts of the control-flow [225].

### 3.2.4 Constant-time

To protect against microarchitectural timing attacks, a solution is to make the execution time of a program and its effect on the microarchitectural state independent from the secret input as illustrated in Figure 3.2. This can be achieved with *constant-time programming* (a.k.a. constant-time policy) [30], which is already widely employed to secure cryptographic implementations (e.g. BearSSL [36], NaCL [41], HACL* [272], etc).

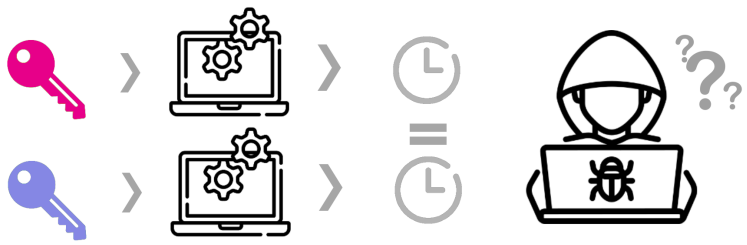


FIGURE 3.2 – Illustration of constant-time programming. Clock made by bqlqn and other icons made by Freepik from www.flaticon.com.

On the one hand, to protect against attacks that can expose information about the sequence of instructions executed by a victim—such as the timing attack on square-and-multiply described in Section 3.2.1, but also microarchitectural attacks targeting the instruction cache, branch prediction units or port contention—constant-time requires the *control-flow* of the victim to be independent from the secret input. On the other hand, to protect against attacks that can expose memory accesses of a victim—typically cache attacks described in Section 3.2.2—constant-time requires the *memory accesses* (including load and store addresses but not their values) to be independent from the secret input. Finally, some versions of constant-time also requires the operands of instructions that execute in non-constant time to be independent from the secret input (e.g. divisions, floating-point operations, etc.). Because this definition of constant-time is architecture-specific, we restrict to a definition that forbids secret-dependent control-flow and memory accesses.

**Definition 6** (Constant-time (CT))**.** *A program is secure w.r.t. constant-time if and only if each pair of executions with the same public input have the same control-flow and memory accesses.*

Note that constant-time is not a property of one execution trace (safety) as it relates *two execution traces* (it is a 2-hypersafety property) and thus requires appropriate tools to efficiently model pairs of traces.

The constant-time programming discipline requires to write a programs in such a way that the control-flow and memory accesses are independent from the secret input. To achieve this, constant-time programming deviates from standard programming behaviors and employs many binary operations to avoid branches, as illustrated in Listing 3.3. For instance, for the square and multiply algorithm (Algorithm 1 in Section 3.2.1), a constant-time implementation should always compute the multiplication, regardless of the value of the secret exponent, and the appropriate value should be selected using branchless code, as implemented in BearSSL [208].

Unfortunately, it is known that constant-time is not necessarily preserved by compilers [231, 43]. For instance, compilers can optimize a branchless code to a code with conditional branches and the `ct_select` function in Listing 3.3 is actually compiled to a conditional branch with `clang-7.1 -m32 -march=i386 -O3`. Writing constant-time programs thus requires a good knowledge of the compiler but this is not always

sufficient: constant-time preservation depends on the version of the compiler and optimization-level [231], forcing developers to resort to manual inspection of binary code.

```
u32_t select(u32_t x, u32_t y, bool bit) {
  return bit ? x : y; // Compiled to a conditional branch
}

u32_t ct_select(u32_t x, u32_t y, bool b) {
  u32_t m = -(u32_t) (((u32_t) b | -(u32_t) b) >> 31);
  /* If b == 0, m is set to 0x0..0 and y is selected */
  /* If b != 0, m is set to 0x1..1 and x is selected */
  return (x & m) | (y & ~m);
}
```

Listing 3.3 – Example of a non-constant-time and a constant-time selection functions where the value of `b` is secret.

**Link with thesis**

In Chapter 4, we develop a binary-level symbolic analyzer for constant-time and generalize it in Chapter 5 to encompass more definitions, including for instance operands of non-constant-time instructions.

## 3.3 Transient execution attacks

**Section overview**

This section first gives an overview of transient execution attacks (cf. Section 3.3.1), then details the two variants of Spectre attacks that we consider in this thesis, namely Spectre-PHT (cf. Section 3.3.2) and Spectre-STL (cf. Section 3.3.3), and their software-based mitigations (cf. Section 3.3.4), some of which we analyze in this thesis. Finally, it defines *speculative constant-time*, the adaptation of constant-time to Spectre attacks, which we analyze in this thesis (cf. Section 3.3.5).

### 3.3.1 Overview of transient execution attacks

Modern processors rely on heavy optimizations to improve performance. To reduce dependencies between instructions and avoid stalling the pipeline when the operands of an instruction are not available, they can execute instructions *out-of-order*. More precisely, instructions are fetched in order and placed in a *reorder buffer* where they can be executed in any order, as soon as their operands are available. On top of this, processors also employ *speculation* mechanisms to predict the outcome of certain instructions before the actual result is known. Instructions streams resulting from a mispeculation—i.e. *transient executions*—are reverted at the architectural level and are meant to be transparent to the program. However, the microarchitectural state that is modified during transient execution is not reverted (e.g. cache state is not restored).

*Spectre attacks* [155] exploit these speculation mechanisms to trigger transient executions of so called *spectre gadgets* that encode secret data in the microarchitectural state, which is finally recovered via microarchitectural attacks (cf. Section 3.2). There are four variants of Spectre attacks, classified according to the speculation mechanism they exploit [65]:

— Spectre-PHT [155, 153] exploits the Pattern History Table, which predicts conditional branches (cf. Section 3.3.2);

— Spectre-BTB [155] exploits the Branch Target Buffer, which predicts branch addresses;

— Spectre-RSB [175, 158] exploits the Return Stack Buffer, which predicts return addresses;

— Spectre-STL [139] exploits the memory disambiguation mechanism, which predicts Store-To-Load dependencies (cf. Section 3.3.3).

Speculation mechanisms at the root of BTB and RSB variants can, in principle, be mistrained to jump to arbitrary addresses [67, 158], In practice, static analyzers cannot precisely model arbitrary jump targets and have to resort to the conservative solution of forbidding indirect jumps. For this reason, we focus on detecting violations of Spectre-PHT and Spectre-STL in this thesis.

### 3.3.2 Spectre-PHT

At conditional instructions, processors can try to predict the value of the condition, via the Pattern History Table (PHT), and *speculatively* execute one branch instead of waiting for the evaluation of the condition. This mechanism is especially useful when the condition of a branch depends on a memory access that is not yet resolved. Instead of waiting for the result and stalling the pipeline, the processor can predict the destination and start executing a branch. When the condition is finally resolved, the processor checks whether its prediction is correct:

— If the prediction is correct, the speculative execution is *committed* and the processor has avoided a pipeline stall, yielding a performance gain;

— If the prediction is incorrect, the speculative execution is *reverted* back to the state before the prediction, with performance similar to a pipeline stall.

In the end, transient execution are transparent to the program (e.g. register values are restored) and only correct execution define the architectural state. However, the *microarchitectural state is not reverted.*

In Spectre-PHT, first introduced as Spectre variant 1 by Kocher et al. [155], the attacker abuses the branch predictor to intentionally mispeculate at a branch. Even if at the architectural level, a conditional statement in a program ensures that memory accesses are within fixed bounds, the attacker can lead the PHT to mispredict the value of a branch to transiently perform an out-of-bound memory access. This out-of-bound access can leave observable effects in the cache that can ultimately be used to recover the out-of-bound read value, as illustrated in Example 5.

> **Example 5** (Spectre-PHT)**.** Consider the program in Listing 3.4 where `idx` is controlled by an attacker. The goal of an attacker is to infer any secret data in `secretarray`—more specifically by encoding secret data into the cache with the function `cache_encode`.
>
> In sequential execution, the conditional branch at line 13 ensures that the array access at line 15 is in-bound, whatever the value of `idx`. Under this condition, the variable `toLeak` can only contain public data loaded from `publicarray`. Therefore,

the program is secure as only public data are encoded into the cache at line 10.

However, in a processor with speculative execution, an attacker can:

1. Mistrain the branch predictor by calling the function `case1` many times with in-bound indexes;
2. Call the function with `idx` set to 131088. The processor will mispredict the conditional branch at line line 13 and transiently execute the array access at line 15. Notice that `publicarray[131088]` aliases with `secretarray[0]` as 131088 = 256 * 512 + 16 = `sizeof(publicarray)` + `sizeof(publicarray2)`. Therefore, `secretarray[0]` is loaded into the variable `toLeak`;
3. The secret data `secretarray[0]` is encoded into the cache at line 10;
4. The processors eventually resolves the conditional and squashes transient instructions;
5. Finally, the attacker can reconstruct the secret from the execution but changes to the microarchitectural state remain. microarchitectural state using cache attacks (cf. Section 3.2.2).

```
1  // Public input
2  uint32_t publicarray_size = 16;
3  uint8_t publicarray[16] = { 1 .. 16 };
4  uint8_t publicarray2[512 * 256];
5  // Secret input
6  uint8_t secretarray[16];
7
8  // This function encodes toLeak in the cache
9  void cache_encode(uint8_t toLeak) {
10   tmp &= publicarray2[toLeak * 512];
11 }
12 void case_1(uint32_t idx) {
13   if(idx < publicarray_size) { // Mispeculated
14     // Out-of-bound read, reads secretarray[0]
15     uint8_t toLeak = publicarray[idx];
16     cache_encode(toLeak);
17   }
18 }
```

Listing 3.4 – Proof-of-concept for Spectre-PHT, taken from Paul Kocher's set of litmus tests [95].

### 3.3.3 Spectre-STL

In out-of-order execution, instructions can be executed out-of-order as soon as their dependencies are resolved. However, determining dependencies between load and store instructions—i.e. Store-to-Load (STL) dependencies—is not an easy task because it requires to precisely determine aliases between loads and stores. A load instruction following a store at the same address cannot be reordered, whereas non-aliasing loads and stores can be reorder. A conservative approach would be to require that loads do not execute before the addresses of all preceding stores have been resolved. However,

modern processors implement more aggressive optimizations: loads can *speculatively bypass* preceding stores. This can lead to incorrect transient execution when the loaded value is overwritten by a store that has been bypassed, but a *memory disambiguation* mechanism detects these cases and revert transient executions to the state preceding the load.

When store instructions are fetched, they are queued in a special microarchitectural buffer called the *store buffer*. In the store-buffer, store instructions can be reordered to avoid stalling on cache-miss stores, and processor can transiently execute store instructions without having to commit changes to the main memory and revert them if transient execution is squashed.

Instead of waiting for preceding stores to be retired, a load instruction can take its value directly from a matching store in the store buffer with *store-to-load forwarding*. Additionally, when the memory disambiguator predicts that a load does not alias with pending stores, it can *speculatively bypass pending stores* in the store buffer and take its value from the main memory [144]. This behavior is exploited in the Spectre-STL [139] variant to load stale values containing secret data that are later encoded in the cache, as illustrated in Example 6.

**Example 6** (Spectre-STL)**.** Consider the program in Listing 3.5 where `idx` is controlled by an attacker. The goal of an attacker is to infer any secret data in `secretarray`. The pointer `ptr` is initialized at line 10 to `secretarray`. The value at `ptr[idx]` is set to 0 at line 12 with a store instruction via a "slow" pointer (i.e. with many indirections).

In sequential execution, the load instruction at line 13 waits for the store at line line 12 to be resolved and loads the value 0. Therefore, the program is secure as only the value 0 can be encoded into the cache at line 13.

However, in transient execution, the load at line 13 speculatively bypasses the slow store at line 12 and loads the value `secretarray[idx]`. Finally, `secretarray[idx]` is encoded into the cache at line 13, and the attacker can recover it later using cache attacks (cf. Section 3.2.2).

```
1  uint8_t publicarray2[512 * 256]; // Public input
2  uint8_t secretarray[16];         // Secret input
3
4  // This function encodes toLeak in the cache
5  void cache_encode(uint8_t toLeak) {
6    tmp &= publicarray2[toLeak * 512];
7  }
8
9  void case_1(uint32_t idx) {
10   uint8_t* ptr = secretarray;
11   uint8_t** slowptr = &ptr;
12   (*slowptr)[idx] = 0;      // Bypassed store
13   cache_encode(ptr[idx]);   // Loads secretarray[idx]
14 }
```

Listing 3.5 – Proof-of-concept for Spectre-STL, taken from https://github.com/IAIK/transientfail/blob/master/pocs/spectre/STL/main.c.

### 3.3.4 Mitigations

Completely disabling speculative execution to protect against Spectre attacks would drastically impact performance and is therefore not a viable option. We give here an overview *software-based defenses* for *Spectre-PHT and Spectre-STL*, which are the most relevant to our work. More defenses, including hardware-based mitigations can be found in a survey on transient execution attacks by Canella et al. [65].

**Link with thesis**

It is not necessary to read this section entirely to understand the rest of this thesis. Only the two following paragraphs on serializing instructions and index-masking are recommended background for Chapter 6. The rest of this section details software-based mitigations for Spectre-PHT and Spectre-STL because some of these mitigations could be analyzed with our tool as future work.

**Serializing instructions** Instead of fully disabling speculations, serializing instructions—such as `lfence` for Intel and `csdb` for arm—can be used to selectively disable speculations. The `lfence` instruction, blocks the execution until preceding instructions have been retired (and until all speculation have been resolved). To protect a conditional instruction against Spectre-PHT, it is sufficient to add an `lfence` instruction before both of its successors. However protecting all conditional branches of a program would be as inefficient as disabling speculations. Consequently, a better approach is to selectively insert fences using static analysis [253, 250]. An example of this is the `/Qspectre` switch in Microsoft's C++ compiler (MSVC) which uses static analysis to selectively insert `lfence` instructions when detecting a vulnerable pattern [197]. However, Paul Kocher [154] showed that Microsoft's analyzer misses many gadgets, which are compiled to unprotected code. Because missed gadgets can compromise the security of the whole program, it is important for these static analyzers to adopt a conservative approach in order to not leave any unprotected gadgets.

To protect against Spectre-STL, it is also possible to insert serializing instructions between stores and loads, or between a load and the disclosure gadget. Because inserting fences between all stores and loads would significantly degrade performance [8], Intel recommends to only apply this mitigation selectively [143].

Arm also provides speculative store bypass barriers (SSBB and PSSBB) to protect against Spectre-STL [142]. They prevent load instructions after the barrier from bypassing store instructions before the barrier.

Finally, `lfence` instructions block speculative execution of instruction after the barrier but they do not block speculative instruction fetches and microarchitectural behavior that occur pre-execution. These speculative instructions fetches can be used to leak information—e.g. by powering up the advanced vector extension (AVX) unit [225].

**Index masking.** Index-masking [113] is a mitigation that has been proposed in Apple's Webkit [113] to protect against Spectre-PHT. The idea is to strengthen conditional bound checks with branchless bound checks. As illustrated in Listing 3.7, a mask is applied on the index before an array access to ensure that the access is always in bounds. The protection is fully effective if the size of the array is a power of two; otherwise, it still limits the range of the out-of-bound access.

8. The closest evaluation of the performance impact is the mitigation for Load-Value-Injection (LVI) attacks, consisting in adding `lfence` instruction after every vulnerable load instruction, which causes significant performance degradation (overhead of a factor 2 to 19) [55]

```
if (index < array_size) {
  x = array[index];
}
```

LISTING 3.6 – Conditional array-bound check where `array_size` is the length of `array`.

```
if (index < array_size) {
  x = array[index & mask];
}
```

LISTING 3.7 – Index masking countermeasure applied to Listing 3.6 where `mask=next_power_of_2(array_size)-1`

**Link with thesis**

In Section 6.6.1 we analyze with our tool a set of small programs protected with the index-masking countermeasure. We show that, while this countermeasure effectively protects against Spectre-PHT, it may introduce Spectre-STL violations.

**Pointer poisoning.** Pointer poisoning is a second Spectre-PHT mitigation strategy that has been implemented in Apple's Webkit [113]. Contrary to index-masking, it does not restrict to array bound checks but can be applied to other conditional checks (like checking the type of an object).

In Webkit, pointers are xored with a pseudo-random poisoned value, specific to the type of the pointed object. To perform a memory access, a poisoned pointer must first be unpoisoned, using the correct poison value. Poison values are chosen such that poisoned pointers (or pointers unpoisoned using the wrong value) are very likely to hit unmapped memory. In case of mispeculation of a branch type check, the wrong poison value is used for unpoisoning the pointer and the memory access fails, preventing the attack.

**Speculative load hardening (SLH).** Speculative-load-hardening [66] (SLH) is a mitigation that has been proposed in Clang, to protect against Spectre-PHT. The idea behind SLH is to introduce a data dependency between the condition and the load index, reducing the chances that the load is speculatively executed. Moreover, even though the load is speculatively executed, the index is zeroed-out in case of mispeculation and thus cannot leak data [9].

An example of SLH is given in Listing 3.8. The load at line 11 can only be executed once the value of `condition` is known, and at this point the speculation on the conditional branch at line 6 can also be resolved. If the branch is mispeculated anyway, the pointer is zeroed at line 10. Another alternative is to harden the loaded value, as illustrated at line 16. SLH requires that the conditional selections of the mask at line 7 and 13 are implemented using branchless and unpredicted conditional updates of registers (which are available in all modern architectures).

**Site isolation** Site isolation [212] has been proposed by Google to reinforce isolation between websites by putting different websites in their own process. In the context of transient execution attacks, where software boundaries between sites can be speculatively bypassed, it limits memory disclosure to a single process, meaning that a malicious website cannot exfiltrate data from other websites.

9. Under the hypothesis that memory addresses that are accessible from hardened pointers do not contain secrets (typically the low 2GB of the address space).

```
uintptr_t all_ones_mask = UINTPTR_MAX;
uintptr_t all_zeros_mask = 0;

void example(int* pointer1, int* pointer2) {
  uintptr_t mask = all_ones_mask;
  if (condition) {
    mask = !condition ? all_zeros_mask : mask;
    [...]
    // Harden the pointer so it cannot be loaded
    pointer1 &= mask;
    leak(*pointer1);
  } else {
    mask = condition ? all_zeros_mask : mask;
    [...]
     // Harden the loaded value
    int value2 = *pointer2 & predicate_state;
    leak(value2);
  }
}
```

Listing 3.8 – Examples of speculative load hardening from https://llvm.org/docs/SpeculativeLoadHardening.html

**Disabling speculative store bypass.** To protect against Spectre-STL, the approach recommended by Intel is to employ Speculative Store Bypass Disable (SSBD) [143]. This mitigation (which requires microcode update) disables the speculative store bypass mechanism and prevents loads from being speculatively executed before the address of preceding stores are resolved. It can be enabled system-wide or for a single process.

**Timer reduction** Exploiting timing side-channel requires access to an accurate timer. Reducing the accuracy of timers makes timing side-channel attacks more challenging, e.g. by preventing an attacker from distinguishing a cache hit from a cache miss. This approach has been adopted in many web browsers [113, 137, 242, 251], to protect against untrusted JavaScript code, by reducing timers precision in JavaScript and adding jitter. However, this technique does not completely prevent side-channel attacks but only makes them more challenging. Moreover, this approach has been shown to be insufficient to prevent microarchitectural timing attacks in browsers as timing information in JavaScript can be obtained from many other sources [224].

### 3.3.5 Speculative constant-time

Defending security-critical programs against Spectre is crucial but is not an easy task: countermeasures such as serialization instructions, index masking, or speculative load hardening must be applied selectively in order to degrade performance, but missing exploitable gadgets can compromise the security of the whole system [154]. While constant-time programming (cf. Section 3.2.4) is sufficient to avoid leaking secrets via timing side-channels in sequential execution, it is not sufficient to prevent Spectre attacks. For example, the program in Listing 3.4 is a trivially constant-time since there is no secret-dependent branch or memory access. However, the program is

vulnerable to Spectre-PHT since an attacker can mistrain the branch predictor and leak secrets in transient execution.

Speculative constant-time [67] is a recent security property that extends constant-time to take the semantics of speculative execution into account.

**Definition 7** (Speculative constant-time (SCT) [67])**.** *A program is secure w.r.t. speculative constant-time if and only if for each pair of (speculative) executions with the same public input and agreeing on their speculation decisions, (e.g. follow regular path or mispeculate at a branch), then their control-flow and memory accesses are equal.*

Note that SCT (like constant-time and other information flow properties) is not a property of one execution trace (safety) as it relates *two execution traces* (it is a 2-hypersafety property) and thus requires appropriate tools to efficiently model pairs of traces.

**Link with thesis**

In Chapter 6, we develop a symbolic analyzer that takes into account the semantics of speculative execution and analyzes speculative constant-time.

# Part III

# Contributions

# Chapter 4

# BINSEC/REL: Symbolic Binary Analyzer for Constant-Time

**Chapter overview**

The constant-time programming discipline is an effective countermeasure against timing side-channel attacks, requiring the control flow and the memory accesses to be independent from the secrets. Yet, writing constant-time code is challenging as it demands to reason about pairs of execution traces (2-hypersafety property) and it is generally not preserved by the compiler, requiring binary-level analysis. Unfortunately, current verification tools for constant-time either reason at higher level (C or LLVM), or sacrifice bug-finding or bounded-verification, or do not scale. We tackle the problem of designing an efficient binary-level verification tool for constant-time providing both bug-finding and bounded-verification.

The technique builds on relational symbolic execution enhanced with new optimizations dedicated to information flow and binary-level analysis, yielding a dramatic improvement over prior work based on symbolic execution. We implement a prototype, BINSEC/REL, and perform extensive experiments on a set of 338 cryptographic implementations, demonstrating the benefits of our approach. Using BINSEC/REL, we also automate a prior manual study on the preservation of constant-time by compilers. Interestingly, we discover that `gcc -O0` and backend passes of `clang` introduce violations of constant-time in implementations that were previously deemed secure by a state-of-the-art constant-time verification tool operating at LLVM level, showing the importance of reasoning at binary-level.

## 4.1 Introduction

Timing channels occur when timing variations in a sequence of events depend on secret data. They can be exploited by an attacker to recover secret information such as plaintext data or secret keys. *Timing attacks*, unlike other side-channel attacks (e.g based on power consumption [157], electromagnetic emissions [209]) do not require special equipment and can be performed remotely [51, 50]. First timing attacks exploited secret-dependent *control flow* with measurable timing differences to recover secret keys from cryptosystems [156]. With the increase of shared architectures (e.g. infrastructure as a service) arise more powerful attacks, where an attacker can monitor the cache of the victim and recover information on secret-dependent *memory*

*accesses* [39, 195, 199]. More generally, *microarchitectural attacks* enable an attacker to observe changes to the microarchitectural state [3, 104, 103, 5, 225, 115].

It is of paramount importance to implement adequate countermeasures to protect cryptographic implementations from these attacks. Simple countermeasures consisting in adding noise or dummy computations can reduce timing variations and make attacks more complex. Yet, these mitigations eventually become vulnerable to new generations of attacks and provide only *pseudo* security [217].

The *constant-time* programming discipline (CT) [30], a.k.a. constant-time policy, is a software-based countermeasure to timing and microarchitectural attacks which requires the control flow and the memory accesses of the program to be independent from the secret input [1]. Constant-time has been proven to protect against cache-based timing attacks [30], making it the most effective countermeasure against timing attacks, already widely used to secure cryptographic implementations (e.g. BearSSL [36], NaCL [41], HACL* [272], etc).

**Problem.** Writing constant-time code is complex as it requires low-level operations deviating from traditional programming behaviors. Moreover, this effort is brittle as it is generally not preserved by compilers [231, 147]. For example, reasoning about constant-time requires to know whether the code `c=(x<y)-1` will be compiled to branchless code, but this depends on the compiler version and optimization [231]. As shown in the attack on a "constant-time" implementation of elliptic curve Curve25519 [147], writing constant-time code is error prone [147, 231, 217].

Several constant-time analysis tools have been proposed to analyze source code [21, 45], or LLVM code [10, 49], but leave the gap opened for violations introduced in the executable code either by the compiler [231] or by closed-source libraries [147].

Binary-level tools for constant-time using dynamic approaches [166, 69, 254, 257] can find bugs but otherwise miss vulnerabilities in unexplored portions of the code, making them incomplete; whereas static approaches [164, 99, 100] cannot report precise counterexamples—making them of minor interest when the implementation cannot be proven secure. Aside from a posteriori analysis, correct-by-design approaches [7, 46, 68, 272] require to reimplement cryptographic primitives from scratch, and OS-based countermeasures [271, 173, 114, 126] incur runtime overhead and require specific OS- or hardware-support.

**Challenges.** Two main challenges arise in the verification of constant-time:

**C1** It is notoriously difficult to adapt formal methods to binary-level because of the lack of structure information (data and control) and the explicit representation of the memory as a large array of bytes [97, 23];

**C2** Common verification methods do not directly apply because information flow properties like constant-time are not regular safety properties but 2-hypersafety properties [77], and their standard reduction to safety on a transformed program via *self-composition* [33] is inefficient [241].

*Symbolic execution* (SE) [121, 63] is a technique that scale well on binary code and has already shown promising results for binary analysis [121, 19, 72, 230, 90, 27, 220]. To address challenge **C1**, we build on symbolic execution tools for binary analysis [96] which provide suitable abstractions for reasoning about binary code (i.e. loader, disassembler, intermediate language, formula simplifications). However, the adaptation of symbolic execution to 2-hypersafety properties through (variants of) self-composition

1. Some versions of constant-time also require that the size of operands of variable-time instructions (e.g. integer division) is independent from secrets.

suffers from a scalability issue [24, 98, 182]. Some recent approaches achieve better scaling, but at the cost of sacrificing either bounded-verification [254, 238] (by doing under-approximations) or bug-finding [49] (by doing over-approximations).

Instead of using self-composition combined with off-the-shelf symbolic analyses, we address challenge **C2** by proposing a symbolic analysis modeling pairs of executions that maximizing sharing between these pairs of executions. This idea, of analyzing pairs of executions in a single symbolic execution instance while maximizing sharing between them, originates from back-to-back testing and has first been coined as *ShadowSE* [62, 196, 159]. It has been adapted later in the context of 2-hypersafety verification and called *relational symbolic execution* (RelSE) [105].

However, a direct adaptation of *RelSE does not scale in the context of binary-level analysis* because of the representation of the memory as a symbolic array. This symbolic array cannot be shared directly between executions, which prevents efficient information flow tracking, and results in sending *a high number of queries* to the constraint solver.

**Proposal.** *We tackle the problem of designing an efficient symbolic verification tool for constant-time at binary-level that leverages the full power of symbolic execution without sacrificing bug-finding nor bounded-verification.* We present BINSEC/REL, the first efficient binary-level automatic tool for bug-finding and bounded-verification of constant-time at binary-level. It is compiler-agnostic, targets x86 and ARM architectures and does not require source code.

The technique is based on *relational symbolic execution* [62, 105]: it models two execution traces following the same path in the same symbolic execution instance and *maximizes sharing between them.* We show via experiments (Section 4.6.3) that RelSE alone does not scale at binary-level to analyze constant-time on real cryptographic implementations. Therefore, we propose dedicated optimizations for *binary-level RelSE.* Our key technical insights are:

1. Complement RelSE with dedicated optimizations offering a fine-grained information flow tracking in the memory, improving sharing at binary-level;
2. Use this sharing to track secret-dependencies and reduce the number of queries sent to the solver.

BINSEC/REL can analyze about 23 million instructions in 98 min (3860 instructions per second), outperforming similar state of the art binary-level verification tools based on symbolic execution [238, 254] (cf. Table 4.9, page 83), while being still correct for both bug-finding and bounded-verification of constant-time.

**Contributions.** Our contributions are the following:

— We design dedicated optimizations for information flow analysis at binary-level. First, we complement relational symbolic execution with a new *on-the-fly* simplification for *binary-level* analysis, to track secret-dependencies and maximize sharing in the memory (Section 4.4.2.1). Second, we design new simplifications for *information flow* analysis: untainting (Section 4.4.2.2) and fault-packing (Section 4.4.2.3). Moreover, we formally prove that our analysis is correct for bug-finding and bounded-verification of constant-time (Section 4.4.3);

— We propose a verification tool named BINSEC/REL for constant-time analysis (Section 4.5). Extensive experimental evaluation (338 samples) against standard approaches based on self-composition and RelSE (Section 4.6.3) shows that it can find bugs in real-world cryptographic implementations much faster than

these techniques (×715 speedup) and can achieve bounded-verification when they time out, with performance close to standard SE (×2 overhead);

— In order to prove the effectiveness of BINSEC/REL, we perform an extensive analysis of constant-time at binary-level. In particular, we analyze 296 cryptographic binary-codes previously verified at a higher-level, including codes from HACL* [272], BearSSL [36], Libsodium [41]; we replay known bugs in 42 samples including Lucky13 [6]; and automatically generate counterexamples (Section 4.6.2);

— Simon *et al.* [231] have demonstrated that `clang` optimizations break constant-timeness of code. We extend this work in four directions going from 192 to 408 configurations (Section 4.6.2):

1. we automatically analyze the code that was manually checked in [231],
2. we add new implementations,
3. we add the `gcc` compiler and a more recent version of `clang`,
4. we add ARM binaries.

Interestingly, we discovered that `gcc -O0` and backend passes of `clang` with `-O3 -m32 -march=i386` introduce violations of constant-time that cannot be detected by LLVM verification tools like `ct-verif` [10].

**Discussion.** Our technique is shown to be highly efficient on bug-finding and bounded-verification compared to alternative approaches, paving the way to a systematic binary-level analysis of constant-time on cryptographic implementations, while our experiments demonstrate the importance of developing constant-time verification tools reasoning at binary-level.

**Related background**

The following background is recommended before reading this chapter:

— introduction of symbolic execution, given in Section 2.2,

— definition of information-flow policies and self-composition, given in Section 3.1,

— definition of constant-time, given in Section 3.2.4,

— the basics of binary analysis, given in Section 2.3—in particular the low-level language used in this section, defined in Section 2.3.4 and the binary-level symbolic execution on which we build, defined in Section 2.3.5.

## 4.2 Motivating example

**Section overview**

This section first illustrate the constant-time property on a small example (cf. Section 4.2.1). Next, it illustrates the standard adaptation of symbolic execution, via self-composition, to constant-time verification and its limitations (cf. Section 4.2.2). To overcome the limitations of self-composition, it introduces relational symbolic execution (cf. Section 4.2.3). Finally, it presents the challenges of combining relational symbolic execution with binary analysis, motivating the need for dedicated optimizations (cf. Section 4.2.4).

### 4.2.1 Constant-time analysis of a toy program

Consider the toy program in Listing 4.1. The constant-time policy considers that the value of the conditional expression at line 3 and the address of the memory access at line 5 are *leaked*. We say that a *leak is insecure* if it depends on the secret input. Conversely, a *leak is secure* if it does not depend on the secret input. Constant-time holds for a program if there is no insecure leak.

```
1 x = secret_input();
2 y = public_input();
3 if (y == 0) { return 0; } // leak y = 0
4 else { z = x / y; }
5 return tab[z];              // leak z
```

Listing 4.1 – Toy program with one control-flow leak and one memory leak.

**Example.** Consider two executions of this program with the same public input: $(x, y)$ and $(x', y')$ where $y = y'$. Intuitively, we can see that the leakages produced at line 3, $y = 0$ and $y' = 0$, are necessarily equal in both executions because $y = y'$; hence this leak does not depend on secret input and is secure. On the contrary, the leakages at line 5 can differ in both executions. For instance, take $y := 1$, $x := 0$ and $x' := 1$, then the leak is 0 in the first execution and 1 in the second; hence this leak depends on secret input and is insecure.

*The goal of an automatic analysis is to prove that the leak at line 3 is secure and to return concrete input showing that the leak at line 5 is insecure.*

### 4.2.2 Symbolic execution and self-composition

Symbolic execution can be adapted for constant-time analysis following the self-composition principle. Instead of self-composing the program, we rather self-compose the formula with a renamed version of itself plus a precondition stating that the low inputs are equal [201]. Basically, this amounts to model *two different executions following the same path and sharing the same low input* in a single formula.

At each conditional statement, *exploration queries* are sent to the solver to determine satisfiable branches—followed by both executions (similar to standard SE exploration). Moreover, additional *insecurity queries* specific to constant-time are sent before each conditional statement and memory access to determine whether they depend on secret—if an insecurity query is satisfiable then a constant-time violation is found.

**Example.** As an illustration, consider the symbolic execution of the program in Listing 4.1 where variables `x` and `y` are assigned symbolic values $x$ and $y$.

At the first conditional (line 3), the symbolic execution generates a formula of the condition: $c \triangleq (y = 0)$. Second, self-composition is applied on the formula with precondition $y = y'$ to constrain the low inputs to be equal in both executions. Finally, a postcondition $c \neq c'$ asks whether the value of the condition can differ, resulting in the following insecurity query:

$$y = y' \ \wedge \ c \triangleq (y = 0) \ \wedge \ c' \triangleq (y' = 0) \ \wedge \ c \neq c'$$

This formula is sent to an SMT solver. If the solver returns UNSAT, meaning that the query is not satisfiable, then the conditional does not differ in both executions and thus is secure. Otherwise, it means that the outcome of the conditional depends on the secret and the solver returns a counterexample satisfying the insecurity query. Here, the SMT solver `z3` [92] answers that the query is unsatisfiable and we can conclude that the leak is secure.

Similarly, at the memory access (line 5), symbolic execution generates a formula to ask whether the value of the memory access can differ in both executions, resulting in the following insecurity query:

$$y = y' \ \wedge \ \begin{pmatrix} z \triangleq x/y \ \wedge \ y \neq 0 \ \wedge \\ z' \triangleq x'/y' \ \wedge \ y' \neq 0 \end{pmatrix} \ \wedge \ z \neq z'$$

Here, the solver answers that the query is satisfiable and returns as a counterexamples a model $\{x = 0;\ x' = 1;\ y = 1;\ y' = 1\}$, for which $z = 0$ and $z' = 1$.

**Limits.** Basic self-composition suffers from two weaknesses:

— It generates many insecurity queries—at each conditional statement and memory access. Yet, in the previous example it is clear that the conditional does not depend on secrets and the query could be spared with better information flow tracking.

— The whole original formula is duplicated, so the size of the self-composed formula is twice the size of the original formula. Yet, because parts of the program that only depend on public input are equal in both executions, the self-composed formula contains redundancies that are not exploited.

### 4.2.3 Relational symbolic execution

Self-composition can be improved by maximizing *sharing* between the pairs of executions [62, 196, 159, 105]. Like self-composition, RelSE models two executions of a program $P$, let us call them $p$ and $p'$. The difference is that RelSE, models both $p$ and $p'$ at the same time. Variables of $P$ are mapped to *relational expressions* which are either *pairs* of expressions $\langle e \,|\, e' \rangle$ or *simple* expressions $\langle e \rangle$. Variables that *must be equal* in $p$ and $p'$—i.e. that only depend on low input—are represented as *simple* expressions; whereas variables that *may be different* (i.e. which may depend on secrets) are represented as *pairs* of expressions. Secret-dependencies are propagated (in a conservative way) through symbolic execution using these relational expressions: if the evaluation of an expression only involves simple operands, its result will be a simple expression; whereas if it involves a pair of expressions, its result will be a pair of expressions. Additionally, notice that because constant-time forbids secret-dependent control-flow, there are no implicit flows (i.e. secret-dependencies resulting from secret-dependent control-flow). This representation offers two main advantages:

— It enables sharing redundant parts of $p$ and $p'$, reducing the size of the final formula;

— Variables that map to simple expressions cannot depend on secret input, which makes it possible to spare insecurity queries.

**Example.** Consider RelSE of the toy program in Listing 4.1. Variable `x` is assigned a pair of expressions $\langle x \,|\, x' \rangle$ and `y` is assigned a simple expression $\langle y \rangle$. Note that the precondition that public variables are equal is now implicit since we use the same

symbolic variable in both executions. At line 3, the conditional expression is evaluated to $c \triangleq \langle y > 0 \rangle$ and we need to check that the leakage of $c$ is secure. Since $c$ maps to a simple expression, we know by definition that it does not depend on the secret, hence we can spare the insecurity query.

*RelSE maximizes sharing between both executions and tracks secret-dependencies enabling to spare insecurity queries and reduce the size of the formula.*

### 4.2.4 Challenge of binary-level analysis

Recall that in binary-level SE, the memory is represented as a special variable of type $(Array\ \mathcal{B}v_{32}\ \mathcal{B}v_8)$. We cannot directly store relational expressions in it, therefore we have to duplicate it in order to store high inputs at the beginning of the execution. Consequently the *memory is always duplicated* and every *select* operation will evaluate to a duplicated expression, preventing sharing and secret-tracking in many situations.

**Example.** As an illustration, consider the compiled version of the previous program, given in Listing 4.2. The steps of RelSE on this program are given in Figure 4.1. When the secret input is stored in memory at line 1, the array representing the memory is duplicated. This propagates to the load expression in `eax` at line 3, and to the conditional expression at line 4. Intuitively, at line 4, `eax` should be equal to the simple expression $\langle \lambda \rangle$ in which case we could spare the insecurity query like in the previous example on source code. However, because dependencies cannot be tracked in the array representing the memory, `eax` evaluates to a pair of *select* expressions and we have to send the insecurity query to the solver.

```
1 store ebp-8 := ⟨x|x'⟩    // store high input on stack
2 store ebp-4 := ⟨y⟩        // store low input on stack
3 eax := load ebp-4         // assign ⟨y⟩ to eax
4 ite eax ? l1 : l2         // leak ⟨y = 0⟩
5 [...]
```

LISTING 4.2 – Compiled version of Listing 4.1, where $\texttt{x} := \langle x \mid x' \rangle$ (resp. $\texttt{x} := \langle x \rangle$) denotes that variable `x` is assigned a high (resp. low) input.

(*init*) `mem` $\mapsto \langle \mu_0 \rangle$ and `ebp` $\mapsto \langle ebp \rangle$
(1) `mem` $\mapsto \langle \mu_1 \,|\, \mu_1' \rangle$ where $\mu_1 \triangleq store(\mu_0, ebp - 8, x)$ and $\mu_1' \triangleq store(\mu_0, ebp - 8, x')$
(2) `mem` $\mapsto \langle \mu_2 \,|\, \mu_2' \rangle$ where $\mu_2 \triangleq store(\mu_1, ebp - 4, y)$ and $\mu_2' \triangleq store(\mu_1', ebp - 4, y)$
(3) `eax` $\mapsto \langle c \,|\, c' \rangle$ where $c \triangleq select(\mu_2, ebp - 4)$ and $c' \triangleq select(\mu_2', ebp - 4)$
(4) *leak* $\langle c \neq 0 \,|\, c' \neq 0 \rangle$

FIGURE 4.1 – RelSE of program in Listing 4.2 where `mem` is the memory variable, `ebp` and `eax` are registers, $\mu_0, \mu_1, \mu_1', \mu_2, \mu_2'$ are symbolic array variables, and $ebp, x, x', y, c, c'$ are symbolic bitvector variables

**Practical impact.** Table 4.1 reports the performance of constant-time analysis on an implementation of elliptic curve Curve25519-donna [40]. *Both self-composition and RelSE fail to prove the program secure in less than 1h. RelSE* does reduce the number of queries compared to self-composition, but it is clearly not sufficient.

| Version | #Instr | #Query | Time | #Instr/s | Status |
|---|---|---|---|---|---|
| *Self-composition* (e.g. [238]) | 11k | 9051 | ⧗ | 3 | ⧗ |
| *RelSE* (e.g. [105]) | 13k | 5486 | ⧗ | 4 | ⧗ |
| BINSEC/REL | 10M | 0 | 1166 | 8576 | ✓ |

TABLE 4.1 – Performance of constant-time analysis on `donna` compiled with `gcc-5.4 -O0`, in terms of number of unrolled instructions explored (#Instr), number of queries (#Query), execution time in seconds (Time), instructions explored per second (#Instr/s), and status (Status) set to secure (✓) or timeout (⧗) set to 3600s.

**Our solution.** To mitigate this issue, we propose dedicated simplifications for binary-level relational symbolic execution, which allow a precise tracking of secret-dependencies *in the memory* (details in Section 4.4.2). In the particular example of Table 4.1, our prototype BINSEC/REL *proves that the code is secure* in less than 20 minutes. Our simplifications simplify all the queries, resulting in a $\times 2000$ speedup compared to standard RelSE and self-composition in terms of number of instructions explored per second.

## 4.3 Concrete semantics and leakage model

**Section overview**

This section, introduce the leakage model (cf. Section 4.3.1) and defines the constant-time property (cf. Section 4.3.2).

### 4.3.1 Leakage model

We define the leakage model in a low-level language called Dynamic Bitvectors Automatas (DBA) [28], introduced in Section 2.3.4. The behavior of the program is modeled with an instrumented operational semantics taken from prior work [35] in which each transition is labeled with an explicit notion of leakage. Along the execution, memory addresses (in $BV_n$) and program locations (in $Loc$) are leaked. A transition from a configuration $c$ to a configuration $c'$ produces a leakage $t \in BV_n \times Loc$, denoted $c \underset{t}{\rightarrow} c'$. Analogously, the evaluation of an expression $e$ in a configuration $(l, r, m)$, produces a leakage $t$, denoted $(l, r, m)\ e \vdash_t bv$. The leakage of a multistep execution is the concatenation of leakages, denoted $\cdot$, produced by individual steps. We use $\underset{t}{\rightarrow}^k$ with $k$ a natural number to denote $k$ steps in the concrete semantics.

The concrete semantics is given in Figure 4.2 for expressions and Figure 4.3 for instructions. Section 2.3.4 gives a functional explanation of the rules, so we only detail here the leakage. Leakage by memory accesses occur during the execution of LOAD and STORE rules where the index is evaluated and its value is leaked. Control-flow leakages occur during the execution of I-JUMP, ITE-TRUE and ITE-FALSE where the next location is evaluated and its value is leaked.

### 4.3.2 Secure program

Let $\mathcal{V}_h \subseteq \mathcal{V}$ be the set of high (secret) variables and $\mathcal{V}_l = \mathcal{V} \setminus \mathcal{V}_h$ be the set of low (public) variables. Analogously, we define $\mathcal{A}_h \subseteq BV_{32}$ (resp. $\mathcal{A}_l = BV_{32} \setminus \mathcal{A}_h$)

**Expr**

$$\text{CST}\ \frac{}{(l,r,m)\ \texttt{bv} \vdash_\varepsilon \texttt{bv}} \qquad \text{VAR}\ \frac{}{(l,r,m)\ \texttt{v} \vdash_\varepsilon r\ \texttt{v}}$$

$$\text{UNOP}\ \frac{(l,r,m)\ e \vdash_t \texttt{bv}}{(l,r,m)\ \blacktriangleleft e \vdash_t \blacktriangleleft \texttt{bv}} \qquad \text{BINOP}\ \frac{(l,r,m)\ e_1 \vdash_{t_1} \texttt{bv}_1 \qquad (l,r,m)\ e_2 \vdash_{t_2} \texttt{bv}_2}{(l,r,m)\ e_1 \blacklozenge e_2 \vdash_{t_1 \cdot t_2} \texttt{bv}_1 \blacklozenge \texttt{bv}_2}$$

$$\text{LOAD}\ \frac{(l,r,m)\ e \vdash_t \texttt{bv}}{(l,r,m)\ \texttt{load}\ e \vdash_{t \cdot \texttt{bv}} m\ \texttt{bv}}$$

FIGURE 4.2 – Concrete evaluation of DBA expressions.

as the addresses containing high (resp. low) input in the initial memory. The *low-equivalence relation* over concrete configurations $c$ and $c'$, denoted $c \simeq_l c'$, is defined as the equality of low variables and low parts of the memory.

**Definition 8** (Low equivalence of states ($c \simeq_l c'$))**.** *Two configurations $c \triangleq (l,r,m)$, and $c' \triangleq (l',r',m')$ are low-equivalent if and only if:*

- *for all variable $v \in \mathcal{V}_l$, $r\ v = r'\ v$ and,*
- *for all address $a \in \mathcal{A}_l$, $m\ a = m'\ a$.*

A program is constant-time up to $k$ if and only if all pairs of low-equivalent initial configurations $c_0$ and $c'_0$ evaluating in $k$ steps to $c_k$ and $c'_k$ produce the same leakage.

**Definition 9** (Constant-time up to $k$)**.** *A program is constant-time (CT) up to $k$ if and only if:*

$$c_0 \simeq_l c'_0 \ \wedge\ c_0 \xrightarrow[t]{}^{k} c_k \ \wedge\ c'_0 \xrightarrow[t']{}^{k} c'_k \implies t = t'$$

*Additionally, a program is constant-time if it is constant-time up to $k$ for all $k \in \mathbb{N}$.*

Notice that, because the leakage determines the control-flow, $t = t'$ implies that $c_k$ and $c'_k$ are at the same program point. Therefore $c_k$ is in a final configuration (i.e. on a `halt` instruction) if and only if $c'_k$ is in a final configuration. Therefore, although it is not explicitly visible, our definition of constant-time is *termination sensitive*.

## 4.4 Binary-level relational symbolic execution

**Section overview**

This section presents the technical contributions of this chapter.

- It introduces our binary-level relational symbolic execution for constant-time analysis (cf. Section 4.4.1);
- It details the optimizations we propose to improve performance of our analysis (cf. Section 4.4.2);
- It presents theorems on our analysis (i.e. bug-finding and bounded-verification) and their proofs (cf. Section 4.4.3).

**Instr**

HALT
$$\frac{\mathbb{P}[l] = \texttt{halt}}{(l, r, m) \to (l, r, m)}$$

S-JUMP
$$\frac{\mathbb{P}[l] = \texttt{goto } l'}{(l, r, m) \to (l', r, m)}$$

I-JUMP
$$\frac{\mathbb{P}[l] = \texttt{goto } e \qquad (l, r, m)\ e \vdash_t \texttt{bv} \qquad l' \triangleq to\_loc(\texttt{bv})}{(l, r, m) \xrightarrow[t \cdot l']{} (l', r, m)}$$

ITE-TRUE
$$\frac{\mathbb{P}[l] = \texttt{ite } e \texttt{ ? } l_1 \texttt{ : } l_2 \qquad (l, r, m)\ e \vdash_t \texttt{bv} \qquad \texttt{bv} \neq 0}{(l, r, m) \xrightarrow[t \cdot l_1]{} (l_1, r, m)}$$

ITE-FALSE
$$\frac{\mathbb{P}[l] = \texttt{ite } e \texttt{ ? } l_1 \texttt{ : } l_2 \qquad (l, r, m)\ e \vdash_t \texttt{bv} \qquad \texttt{bv} = 0}{(l, r, m) \xrightarrow[t \cdot l_2]{} (l_2, r, m)}$$

ASSIGN
$$\frac{\mathbb{P}[l] = \texttt{v := } e \qquad (l, r, m)\ e \vdash_t \texttt{bv}}{(l, r, m) \xrightarrow[t]{} (l + 1, r[\texttt{v} \mapsto \texttt{bv}], m)}$$

STORE
$$\frac{\mathbb{P}[l] = \texttt{store} \quad e := e' \qquad (l, r, m)\ e \vdash_t \texttt{bv} \qquad (l, r, m)\ e' \vdash_{t'} \texttt{bv}'}{(l, r, m) \xrightarrow[t' \cdot t \cdot \texttt{bv}]{} (l + 1, r, m[\texttt{bv} \mapsto \texttt{bv'}])}$$

FIGURE 4.3 – Concrete evaluation of DBA instructions.

### 4.4.1 Binary-level RelSE for constant-time

**Reminder (cf. Section 2.3.5)**

Binary-level symbolic execution relies on the quantifier-free theory of fixed-size bitvectors and arrays (QF_ABV [29]). We let $\Phi$ denote the set of symbolic expressions in the QF_ABV logic and $\varphi$, $\phi$, $\psi$, $\iota$ be symbolic expressions ranging over $\Phi$.

A model $M$ assigns concrete values to symbolic variables. The satisfiability of a formula $\pi$ with a model $M$ is denoted $M \vDash \pi$. In the implementation, an SMT solver is used to determine satisfiability of a formula and obtain a satisfying model, denoted $M \vDash_{\text{SMT}} \pi$. Whenever the model is not needed for our purposes, we leave it implicit and simply write $\vDash \pi$ or $\vDash_{\text{SMT}} \pi$ for satisfiability.

In this section, we define *binary-level relational symbolic execution* for constant-time as an extension of the binary-level symbolic execution presented in Section 2.3.5. A *relational* expression $\widehat{\varphi}$ is either a simple symbolic expression $\langle\varphi\rangle$ or a pair $\langle\varphi_l \,|\, \varphi_r\rangle$ of two symbolic expressions in $\Phi$. We denote $\widehat{\varphi}_{|l}$ (resp. $\widehat{\varphi}_{|r}$), the projection on the left (resp. right) value of $\widehat{\varphi}$. If $\widehat{\varphi} = \langle\varphi\rangle$, then $\widehat{\varphi}_{|l}$ and $\widehat{\varphi}_{|r}$ are both defined as $\varphi$. Let $\boldsymbol{\Phi}$ be the set of relational formulas and $\boldsymbol{\mathcal{B}v_n}$ be the set of relational symbolic bitvectors of size $n$.

#### 4.4.1.1 Security evaluation

For the security evaluation, we define a predicate $secLeak : \mathbf{\Phi} \times \Phi \to Bool$, which ensures that a relational formula does not differ in its right and left components, meaning that it can be leaked securely:

$$secLeak(\widehat{\varphi}, \pi) = \begin{cases} true & \text{if } \widehat{\varphi} = \langle \varphi \rangle \\ true & \text{if } \widehat{\varphi} = \langle \varphi_l \,|\, \varphi_r \rangle \wedge \not\models_{\text{SMT}} \pi \wedge \varphi_l \neq \varphi_r \\ false & \text{otherwise} \end{cases}$$

By definition, a simple expression $\langle \varphi \rangle$ does not depend on secrets and can be leaked securely. Thus it *spares an insecurity query* to the solver. However, a duplicated expression $\langle \varphi_l \,|\, \varphi_r \rangle$ *may* depend on secrets. Hence *an insecurity query must be sent to the solver* to ensure that the leak is secure.

#### 4.4.1.2 Symbolic configuration

Our symbolic evaluation restricts to pairs of traces following the same path—which is sufficient for constant-time because two low-equivalent executions must have the same control-flow. Therefore, a symbolic configuration only needs to consider a single program location $l \in Loc$ at any point of the execution. A *symbolic configuration* is of the form $\left(l, \rho, \widehat{\mu}, \pi\right)$ where:

— $l \in Loc$ is the current program point,

— $\rho : \mathcal{V} \to \mathbf{\Phi}$ is a symbolic register map, mapping variables from a set $\mathcal{V}$ to their symbolic representation as a relational expression in $\mathbf{\Phi}$,

— $\widehat{\mu} : (Array\ \mathcal{B}v_{32}\ \mathcal{B}v_8) \times (Array\ \mathcal{B}v_{32}\ \mathcal{B}v_8)$ is the symbolic memory—a pair of arrays of values in $\mathcal{B}v_8$ indexed by addresses in $\mathcal{B}v_{32}$,

— $\pi \in \Phi$ is the path predicate—a conjunction of conditional statements and assignments encountered along a path.

The location $l$ is not needed for symbolic evaluation of expressions, therefore, we omit it and write $\left(\rho, \mu, \pi\right)$ to denote a symbolic configuration in this context.

#### 4.4.1.3 Symbolic evaluation

Symbolic evaluation of an expression *expr* in a configuration $\left(\rho, \widehat{\mu}, \pi\right)$ to a relational formula $\widehat{\varphi}$, is denoted $\left(\rho, \widehat{\mu}, \pi\right)\ expr \vdash \widehat{\varphi}$ and is given in Figure 4.4 [2]. Detailed explanations of the rules follow:

— CST is the evaluation of a constant `bv` and returns the corresponding symbolic bitvector *bv* as a simple expression;

— VAR is the evaluation of a variable `v` and returns the value mapped to `v` in the register map $\rho$;

— UNOP is the evaluation of a unary operator $\blacktriangleleft\ e$. It evaluates the expression $e$ to a symbolic value $\widehat{\varphi}$, and returns the application of the symbolic operator $\triangleleft$ (corresponding to the concrete operator $\blacktriangleleft$) to $\widehat{\varphi}$. If $\widehat{\varphi}$ is a pair of expression, $\triangleleft$ is trivially lifted to pairs of expressions. The case BINOP, for binary operators, is analogous;

2. This is an adaptation of the rules presented in Figure 2.6, to relational expressions. While some explanations are similar, we still include them here for clarity.

— LOAD is the evaluation of a load expression. The rule computes the symbolic index $\hat{\iota}$, and ensures that it can be leaked securely, i.e. $secLeak(\hat{\iota}, \pi)$ is $true$. It returns a pair of logical *select* formulas from the pair of symbolic memories $\widehat{\mu}$ (the box in the hypotheses should be ignored for now, it will be explained in Section 4.4.2). Note that the returned expression is *always duplicated* as the *select* must be performed in the left and right memories independently.

**Expr**

$$\text{CST}\ \frac{}{(\rho, \widehat{\mu}, \pi)\ \texttt{bv} \vdash \langle bv \rangle} \qquad \text{VAR}\ \frac{}{(\rho, \widehat{\mu}, \pi)\ \texttt{v} \vdash \rho\ \texttt{v}}$$

$$\text{UNOP}\ \frac{(\rho, \widehat{\mu}, \pi)\ e \vdash \widehat{\phi} \qquad \widehat{\varphi} \triangleq \triangleleft\, \widehat{\phi}}{(\rho, \widehat{\mu}, \pi) \blacktriangleleft e \vdash \widehat{\varphi}}$$

$$\text{BINOP}\ \frac{(\rho, \widehat{\mu}, \pi)\ e_1 \vdash \widehat{\phi} \qquad (\rho, \widehat{\mu}, \pi)\ e_2 \vdash \widehat{\psi} \qquad \widehat{\varphi} \triangleq \widehat{\phi} \diamond \widehat{\psi}}{(\rho, \widehat{\mu}, \pi)\ e_1 \blacklozenge e_2 \vdash \widehat{\varphi}}$$

$$\text{LOAD}\ \frac{(\rho, \widehat{\mu}, \pi)\ e_{idx} \vdash \hat{\iota} \qquad \boxed{\widehat{\varphi} \triangleq \langle select(\widehat{\mu}_{|l}, \hat{\iota}_{|l}) \,|\, select(\widehat{\mu}_{|r}, \hat{\iota}_{|r}) \rangle} \qquad secLeak(\hat{\iota}, \pi)}{(\rho, \widehat{\mu}, \pi)\ \texttt{load}\ e_{idx} \vdash \widehat{\varphi}}$$

FIGURE 4.4 – Symbolic evaluation of DBA expressions where $\triangleleft$ (resp. $\diamond$) is the logical counterpart of the concrete operator $\blacktriangleleft$ (resp. $\blacklozenge$).

Symbolic evaluation of instructions, denoted $s \rightsquigarrow s'$ where $s$ and $s'$ are symbolic configurations, is given in Figure 4.5 [3]. Detailed explanations of rules follow:

— S-JUMP is the evaluation of a static jump. It simply moves control to the next target;

— I-JUMP is the evaluation of an indirect jump. The rule first evaluates the jump target to a symbolic expression $\widehat{\varphi}$. Then, it ensures that the jump target can be leaked securely, i.e. $secLeak(\widehat{\varphi}, \pi)$ is $true$. Finally, it finds a concrete value $l'$ for the jump target that satisfies the path predicate, and updates the path predicate and the next location accordingly. Note that this rule is nondeterministic as $l'$ can be any concrete value satisfying the constraint. In practice, we call the solver to enumerate jump targets up to a given bound and continue the execution along valid targets (which jump to an executable section) [4];

— ITE-TRUE is the evaluation of a conditional jump when the expression evaluates to $true$ (the $false$ case is analogous). The rule first evaluates the condition to a symbolic expression $\widehat{\varphi}$. Then it ensures that the truth value of the condition can be leaked securely, (i.e. $secLeak(\boldsymbol{eq_0}\ \widehat{\varphi}, \pi)$ is $true$). If the condition guarding the $true$-branch is satisfiable, the rule updates the path predicate and the next location to explore it;

3. This is an adaptation of the rules presented in Figure 2.7, to relational symbolic execution. While some explanations are similar, we still include them here for clarity.

4. In practice enumerating jump targets up to a given bound might lead to unexplored program paths and consequently missed violations. Therefore our analysis detects and records incomplete jump target enumerations and, if it cannot find any vulnerabilities, it returns "unknown" instead of "secure". Notice that in cryptographic code, indirect jumps usually have a single (or few) target and thus we did not encounter such incomplete enumerations in our experiments.

— ASSIGN is the evaluation of an assignment. It allocates a fresh symbolic variable to avoid term-size explosion, and updates the register map and the path predicate. The content of the box in the hypothesis and the rule CANONICAL-ASSIGN should be ignored for now and will be explained in Section 4.4.2;

— STORE is the evaluation of a store instruction. The rule evaluates the index ($\widehat{\imath}$) and value of the store, and ensures that the index can be leaked securely, i.e. $secLeak(\widehat{\imath}, \pi)$ is $true$. Finally, it updates the symbolic memories and the path predicate with a logical $store$ operation.

**Instr**

$$\text{S\_JUMP}\ \frac{\mathbb{P}[l] = \texttt{goto}\ l'}{(l, \rho, \widehat{\mu}, \pi) \rightsquigarrow (l', \rho, \widehat{\mu}, \pi)}$$

$$\text{I\_JUMP}\ \frac{\begin{array}{c}\mathbb{P}[l] = \texttt{goto}\ e \\ (\rho, \widehat{\mu}, \pi)\ e \vdash \widehat{\varphi} \qquad M \vDash_{\text{SMT}} \pi \wedge \widehat{\varphi}_{|l} = \widehat{\varphi}_{|r} \qquad l' \triangleq to\_loc(M(\widehat{\varphi}_{|l})) \\ \pi' \triangleq \pi \wedge (\widehat{\varphi}_{|l} = \widehat{\varphi}_{|r} = M(\widehat{\varphi}_{|l})) \qquad secLeak(\widehat{\varphi}, \pi)\end{array}}{(l, \rho, \widehat{\mu}, \pi) \rightsquigarrow (l', \rho, \widehat{\mu}, \pi')}$$

ITE-TRUE

$$\frac{\begin{array}{c}\mathbb{P}[l] = \texttt{ite}\ e\ \texttt{?}\ l_{true}\ \texttt{:}\ l_{false} \qquad (\rho, \widehat{\mu}, \pi)\ e \vdash \widehat{\varphi} \\ \pi' \triangleq \pi \wedge (\widehat{\varphi}_{|l} \neq 0) \wedge (\widehat{\varphi}_{|r} \neq 0) \qquad secLeak(\boldsymbol{eq_0}\ \widehat{\varphi}, \pi) \qquad \vDash_{\text{SMT}} \pi'\end{array}}{(l, \rho, \widehat{\mu}, \pi) \rightsquigarrow (l_{true}, \rho, \widehat{\mu}, \pi')}$$

ITE-FALSE

$$\frac{\begin{array}{c}\mathbb{P}[l] = \texttt{ite}\ e\ \texttt{?}\ l_{true}\ \texttt{:}\ l_{false} \qquad (\rho, \widehat{\mu}, \pi)\ e \vdash \widehat{\varphi} \\ \pi' \triangleq \pi \wedge (\widehat{\varphi}_{|l} = 0) \wedge (\widehat{\varphi}_{|r} = 0) \qquad secLeak(\boldsymbol{eq_0}\ \widehat{\varphi}, \pi) \qquad \vDash_{\text{SMT}} \pi'\end{array}}{(l, \rho, \widehat{\mu}, \pi) \rightsquigarrow (l_{false}, \rho, \widehat{\mu}, \pi')}$$

$$\text{ASSIGN}\ \frac{\begin{array}{c}\mathbb{P}[l] = \texttt{v := } e \qquad (\rho, \widehat{\mu}, \pi)\ e \vdash \widehat{\varphi} \qquad \boxed{\neg canonical(\widehat{\varphi})} \\ \widehat{\varphi}' \triangleq fresh(\widehat{\varphi}) \qquad \rho' \triangleq \rho[v \mapsto \widehat{\varphi}'] \qquad \pi' \triangleq \pi \wedge (\widehat{\varphi}'_{|l} = \widehat{\varphi}_{|l}) \wedge (\widehat{\varphi}'_{|r} = \widehat{\varphi}_{|r})\end{array}}{(l, \rho, \widehat{\mu}, \pi) \rightsquigarrow (l+1, \rho', \widehat{\mu}, \pi')}$$

$$\text{CANONICAL-ASSIGN}\ \frac{\begin{array}{c}\mathbb{P}[l] = \texttt{v := } e \\ (\rho, \widehat{\mu}, \pi)\ e \vdash \widehat{\varphi} \qquad canonical(\widehat{\varphi}) \qquad \rho' \triangleq \rho[v \mapsto \widehat{\varphi}]\end{array}}{(l, \rho, \widehat{\mu}, \pi) \rightsquigarrow (l+1, \rho', \widehat{\mu}, \pi)}$$

STORE

$$\frac{\begin{array}{c}\mathbb{P}[l] = \texttt{store}\ e_{idx}\ e_{val} \qquad (\rho, \widehat{\mu}, \pi)\ e_{idx} \vdash \widehat{\imath} \\ (\rho, \widehat{\mu}, \pi)\ e_{val} \vdash \widehat{\nu} \qquad \widehat{\mu}' \triangleq \langle store(\widehat{\mu}_{|l}, \widehat{\imath}_{|l}, \widehat{\nu}_{|l}) \,|\, store(\widehat{\mu}_{|r}, \widehat{\imath}_{|r}, \widehat{\nu}_{|r}) \rangle \\ \pi' \triangleq \pi \wedge \widehat{\mu}'_{|l} = store(\widehat{\mu}_{|l}, \widehat{\imath}_{|l}, \widehat{\nu}_{|l}) \wedge \widehat{\mu}'_{|r} = store(\widehat{\mu}_{|r}, \widehat{\imath}_{|r}, \widehat{\nu}_{|r}) \qquad secLeak(\widehat{\imath}, \pi)\end{array}}{(l, \rho, \widehat{\mu}, \pi) \rightsquigarrow (l+1, \rho, \widehat{\mu}', \pi')}$$

FIGURE 4.5 – Symbolic evaluation of DBA instructions and expressions where $fresh(\widehat{\varphi})$ returns a pair of fresh symbolic variables if $\widehat{\varphi}$ is a pair, or a simple fresh symbolic variable if $\widehat{\varphi}$ is simple; and where $eq_0$ returns true if and only if $x = 0$ and $\boldsymbol{eq_0}$ is the lifting of $eq_0$ to relational formulas.

#### 4.4.1.4 Specification of high and low input

By default, the content of the memory and registers is low so the user has to specify memory addresses that initially contain secret inputs. In our implementation, we provide several ways of specifying secrets that are detailed in Appendix B.2. Addresses of high variables can be specified as offsets from the initial stack pointer `esp`—however, this requires manual reverse engineering. Alternatively, secrets can be specified at source level using dummy functions—which is easier but only applies to libraries or requires access to source code. Finally, a user can specify ELF symbols that are considered secret (when applicable).

#### 4.4.1.5 Bug-finding

A vulnerability is found when the function $\mathit{secLeak}(\widehat{\varphi}, \pi)$ evaluates to *false*. In this case, the insecurity query is satisfiable and the solver returns a model $M$ such that $M \vDash_{\text{SMT}} \pi \wedge (\widehat{\varphi}_{|l} \neq \widehat{\varphi}_{|r})$. The model $M$ assigns concrete values to variables that satisfy the insecurity query. Therefore it can be returned as a concrete counterexample that triggers the vulnerability, along with the current location of the vulnerability.

### 4.4.2 Optimizations for binary-level RelSE

Relational symbolic execution does not scale in the context of binary-level analysis (see *RelSE* in Table 4.6). In order to achieve better scalability, we enrich our analysis with an optimization, called *on-the-fly-read-over-write* (*FlyRow* in Table 4.7), based on *read-over-write* [107]. This optimization simplifies expressions and resolves load operations ahead of the solver, often avoiding to resort to the duplicated memory, and makes it possible to spare insecurity queries. We also enrich our analysis with two further optimizations, called *untainting* and *fault-packing* (*Unt* and *FP* in Table 4.7), specifically targeting RelSE for information flow analysis.

#### 4.4.2.1 On-the-fly read-over-write

Solver calls are the main bottleneck of symbolic execution, and reasoning about *store* and *select* operations in arrays is particularly challenging [107]. Read-over-write (Row) [107] is a simplification for the theory of arrays that efficiently resolves *select* operations. It is particularly efficient in the context of binary-level analysis where the memory is represented as an array and formulas contain many *store* and *select* operations. The standard read-over-write optimization [107] has been implemented as a solver-pre-processing, simplifying a formula before sending it to the solver. While it has proven to be very efficient to simplify individual formulas of a single execution [107], we show in Section 4.6.3.3 that it does not scale in the context of relational reasoning, where formulas model two executions and a lot of queries are sent to the solver.

Thereby, we introduce *on-the-fly read-over-write* (*FlyRow*) to track secret dependencies in the memory and spare insecurity queries in the context of information flow analysis. By keeping track of *relational store expressions* along the execution, it can resolve *select* operations—often avoiding to resort to the duplicated memory—and drastically reduces the number of queries sent to the solver, improving the performance of the analysis.

**Lookup.** The symbolic memory can be seen as the history of the successive *store* operations beginning with the initial memory $\mu_0$. Therefore, a memory *select* can be

resolved by going back up the history and comparing the index to load, with indexes previously stored. FlyRow consists in replacing selection in the memory (Figure 4.4, LOAD rule, boxed hypothesis) by a new function:

$$lookup : ((Array\ \mathcal{B}v_{32}\ \mathcal{B}v_8) \times (Array\ \mathcal{B}v_{32}\ \mathcal{B}v_8)) \times \boldsymbol{\mathcal{B}v_{32}} \rightarrow \boldsymbol{\mathcal{B}v_8}$$

which takes a relational memory and a relational index, and returns the relational bitvector value stored at that index. For simplicity we define the function for simple indexes but it can easily be lifted to relational indexes:

$$\begin{aligned}
lookup(\widehat{\mu}_0, \iota) &= \langle select(\widehat{\mu}_{0|l}, \iota) \,|\, select(\widehat{\mu}_{0|r}, \iota)\rangle \\
lookup(\widehat{\mu}_n, \iota) &= \begin{cases}
\langle \varphi_l \rangle & \text{if } eq^{\#}(\iota,\kappa) \wedge eq^{\#}(\varphi_l, \varphi_r) \\
\langle \varphi_l \,|\, \varphi_r \rangle & \text{if } eq^{\#}(\iota,\kappa) \wedge \neg eq^{\#}(\varphi_l, \varphi_r) \\
lookup(\widehat{\mu}_{n-1}, \iota) & \text{if } \neg eq^{\#}(\iota,\kappa) \\
\langle select(\widehat{\mu}_{n|l}, \iota) \,|\, select(\widehat{\mu}_{n|r}, \iota)\rangle & \text{if } eq^{\#}(\iota,\kappa) = \bot
\end{cases} \\
\text{where } \widehat{\mu}_n &\triangleq \langle store(\widehat{\mu}_{n-1|l}, \kappa, \varphi_l) \,|\, store(\widehat{\mu}_{n-1|r}, \kappa, \varphi_r)\rangle
\end{aligned}$$

where $eq^{\#}(\iota,\kappa)$ is a comparison function relying on *syntactic term equality*, which returns true (resp. false) only if $\iota$ and $\kappa$ are equal (resp. different) in any interpretation. The syntactic equality check is easy to compute—which is crucial for performance of the *lookup* function—but might fail to conclude to (dis-)equality. If the terms $\iota$ and $\kappa$ are not comparable, $eq^{\#}(\iota,\kappa)$ is undefined, denoted $\bot$. Example 7 illustrates the behavior of the *lookup* function.

**Example 7** (Lookup)**.** Let us consider the memory:

$$\widehat{\mu} \quad = \quad \boxed{ebp-4}\boxed{\langle\lambda\rangle} \longrightarrow \boxed{ebp-8}\boxed{\langle\beta \,|\, \beta'\rangle} \longrightarrow \boxed{esp}\boxed{\langle ebp\rangle} \longrightarrow [\,]$$

- A call to $lookup(\widehat{\mu}, ebp-4)$ returns $\lambda$;
- A call to $lookup(\widehat{\mu}, ebp-8)$ first compares the indexes $(ebp-4)$ and $(ebp-8)$. Because it can determine that these indexes are *syntactically distinct*, the function moves to the second element, determines the syntactic equality of indexes and returns $\langle\beta \,|\, \beta'\rangle$;
- A call to $lookup(\widehat{\mu}, esp)$ tries to compare the indexes $(ebp-4)$ and $(esp)$. Without further information, the equality or disequality of *ebp* and *esp* cannot be determined, therefore the lookup is aborted and the *select* operation cannot be simplified.

**Term rewriting.** To improve the conclusiveness of syntactic equality checks for read-over-write, the terms are assumed to be in *normalized* form $\beta + o$ where $\beta$ is a base (i.e. an expression on symbolic variables) and $o$ is a constant offset. The comparison of two terms $\beta + o$ and $\beta' + o'$ in normalized form can be efficiently computed as follows:

- if the bases $\beta$ and $\beta'$ are syntactically equal, then return $o = o'$,
- otherwise the terms are not comparable.

In order to apply *FlyRow*, we normalize all the formulas created during the symbolic execution using *rewriting rules* similar as those defined in [107]. An excerpt of these

rules is given in Figure 4.6. Intuitively, these rewriting rules put symbolic variables at the beginning of the term and the constants at the end (see Example 8).

$$\begin{aligned}
\mathit{normalize}\ (c + t) &= t + c \\
\mathit{normalize}\ (-(t + c)) &= (-t) - c \\
\mathit{normalize}\ ((t + c) + t') &= (t + t') + c \\
\mathit{normalize}\ ((t + c) + c') &= t + (c + c') \\
\mathit{normalize}\ ((t + c) + (t' + c')) &= (t + t') + (c + c') \\
\mathit{normalize}\ ((t + c) - (t' + c')) &= (t - t') + (c - c')
\end{aligned}$$

FIGURE 4.6 – Rewriting rules for normalization (non exhaustive). All expressions belong to the set $\mathcal{B}v$ where $c, c'$ are bitvector constants and $t, t'$ are arbitrary bitvector terms. Note that $(c + c')$ and $c - c'$ are constant values, not terms.

**Example 8** (Normalized formula)**.** *normalize* $((eax + 4) + (ebx + 4)) = (eax + ebx) + 8$

In order to increase the conclusiveness of *FlyRow*, we also need *variable inlining*. However, inlining all variables is not a viable option as it would lead to an exponential term size growth. Instead, we define a *canonical form* $v + o$, where $v$ is a bitvector variable, and $o$ is a constant bitvector offset, and we only inline formulas that are in canonical form (see rule CANONICAL-ASSIGN in Figure 4.5). It enables rewriting of most of the memory accesses on the stack, which are of the form `ebp + o`, while avoiding term-size explosion.

#### 4.4.2.2 Untainting

After the evaluation of a rule with the predicate *secLeak* for a duplicated expression $\langle \varphi_l \,|\, \varphi_r \rangle$, we know that the equality $\varphi_l = \varphi_r$ holds in the current configuration. From this equality, we can deduce useful information about variables that must be equal in both executions. We can then propagate this information to the register map and memory in order to spare subsequent insecurity queries concerning these variables. An example is given in Example 9.

**Example 9** (Untainting)**.** For instance, consider the leak of the duplicated expression $\langle x_l + 1 \,|\, x_r + 1 \rangle$, where $x_l$ and $x_r$ are symbolic variables. If the leak is secure, we can deduce that $x_l = x_r$ and replace all occurrences of $x_r$ by $x_l$ in the rest of the symbolic execution.

We define a function $\mathit{untaint}(\rho, \widehat{\mu}, \widehat{\varphi})$ which takes a register map $\rho$, a memory $\widehat{\mu}$, and a duplicated expression $\widehat{\varphi}$. It applies the rules defined in Figure 4.7 which deduce variable equalities from $\widehat{\varphi}$, propagate them in $\rho$ and $\widehat{\mu}$, and returns a pair of updated register map and memory $(\rho', \widehat{\mu}')$.

This function is used during symbolic execution whenever $\mathit{secLeak}(\widehat{\varphi}, \pi)$ returns true and the expression $\widehat{\varphi}$ is a pair. Intuitively, if the equality of variables $x_l$ and $x_r$ can be deduced from $\widehat{\varphi}_{|l} = \widehat{\varphi}_{|r}$, the *untaint* function replaces occurrences of $x_r$ by $x_l$ in the memory and the register map. As a result, subsequent pairs of expression $\langle x_l \,|\, x_r \rangle$ are replaced by a simple expression $\langle x_l \rangle$ in the rest of the execution [5].

5. We implement untainting with a cache of "untainted variables" that are substituted in the program copy during symbolic evaluation of expressions.

$$
\begin{aligned}
untaint(\rho, \widehat{\mu}, \langle x_l \,|\, x_r \rangle) &= (\rho[x_r \backslash x_l], \widehat{\mu}[x_r \backslash x_l]) \\
\left.\begin{aligned}
untaint(\rho, \widehat{\mu}, \langle \neg t_l \,|\, \neg t_r \rangle) \\
untaint(\rho, \widehat{\mu}, \langle -t_l \,|\, -t_r \rangle) \\
untaint(\rho, \widehat{\mu}, \langle t_l + k \,|\, t_r + k \rangle) \\
untaint(\rho, \widehat{\mu}, \langle t_l - k \,|\, t_r - k \rangle) \\
untaint(\rho, \widehat{\mu}, \langle t_l :: l \,|\, t_r :: l \rangle)
\end{aligned}\right\} &= untaint(\rho, \widehat{\mu}, \langle t_l \,|\, t_r \rangle)
\end{aligned}
$$

FIGURE 4.7 – Untainting rules where $x_l, x_r$ are bitvector variables of the same size, $t_l, t_r, k, l$ are bitvector terms such that $t_l, t_r, k$ have the same size, :: indicates the concatenation of bitvectors, and $f[x_r \backslash x_l]$ indicates that the variable $x_r$ is substituted with $x_l$ in $f$.

#### 4.4.2.3 Fault-packing

Symbolic evaluation generates a large number of insecurity checks for constant-time because of the frequent memory operations. The fault-packing ($FP$) optimization gathers these insecurity checks along a path and postpones their resolution to the end of the basic block (i.e. until the next control-flow-altering statement). An illustration is given in Example 10

**Example 10** (Fault-packing)**.** Consider a basic-block with a path predicate $\pi$. If there are two memory accesses along the basic block that evaluate to $\langle \varphi_{|l} \,|\, \varphi_{|r} \rangle$ and $\langle \phi_{|l} \,|\, \phi_{|r} \rangle$, we would normally generate two insecurity queries $(\pi \wedge \varphi_{|l} \neq \varphi_{|r})$ and $(\pi \wedge \phi_{|l} \neq \phi_{|r})$—one for each memory access. Fault-packing regroups these checks into a single query $\big(\pi \wedge ((\varphi_{|l} \neq \varphi_{|r}) \vee (\phi_{|l} \neq \phi_{|r}))\big)$ sent to the solver at the end of the basic block.

This optimization reduces the number of insecurity queries sent to the solver and thus helps improving performance. However it degrades the precision of the counterexample: checking each instruction individually precisely points to vulnerable instructions, whereas fault-packing reduces accuracy to vulnerable basic blocks only.

Note that even though disjunctive constraints are usually harder to solve than pure conjunctive constraints, those introduced by $FP$ are very simple—they are all evaluated under the same path predicate and are not nested. Therefore, they never end up in a performance degradation (see Table 4.7).

### 4.4.3 Theorems

**Section overview**

This section defines and proves properties of our binary-level RelSE:

— Section 4.4.3.1 claims the correctness of our symbolic execution i.e. each symbolic execution corresponds to a valid pair of concrete executions (no over-approximation);

— Section 4.4.3.3 claims the completeness of our symbolic execution for constant-time programs [a] i.e. for each pair of concrete executions, there exists a corresponding symbolic execution (no under-approximation).

Our main results are the following:

— Section 4.4.3.2 claims that our analysis is correct for bug-finding i.e. when symbolic execution gets stuck, the program is not constant-time;

— Section 4.4.3.4 claims that our analysis is correct for bounded-verification i.e. if symbolic execution does not get stuck up to $k$, the program is constant-time up to $k$.

*a.* Completeness only applies to constant-time programs because our symbolic execution blocks on errors.

**Propositions and Hypothesis.** In order to define properties of our symbolic execution, we use $\rightarrow^k$ (resp. $\rightsquigarrow^k$), with $k$ a natural number, to denote $k$ steps in the concrete (resp. symbolic) evaluation.

**Proposition 1.** *If a program $\mathbb{P}$ is constant-time up to $k$ then for all $i \leq k$, $\mathbb{P}$ is constant-time up to $i$.*

**Proposition 2.** *Concrete semantics is deterministic, c.f. rules of the concrete semantics in Figure 5.1.*

**Hypothesis 1.** *Through this section we assume that theory QF_ABV is correct and complete w.r.t. our concrete evaluation.*

The satisfiability problem for the theory QF_ABV is decidable [185]. Therefore we make the following hypothesis on the solver:

**Hypothesis 2.** *We suppose that the SMT solver for QF_ABV is correct, complete and always terminates. Therefore for a QF_ABV formula $\varphi$, $M \vDash \varphi \iff M \vDash_{\text{SMT}} \varphi$.*

**Hypothesis 3.** *We assume that the program $\mathbb{P}$ is defined on all locations computed during the symbolic execution—notably by the function to_loc in rule* I-JUMP*. Under this hypothesis, and because the solver always terminates (Hypothesis 2), symbolic execution is stuck if and only if an expression $\widehat{\varphi}$ is leaked such that $\text{secLeak}(\widehat{\varphi}, \pi)$ evaluates to $false$. In this case, the solver returns a model $M$ such that $M \vDash \pi \wedge (\widehat{\varphi}_{|l} \neq \widehat{\varphi}_{|r})$ (from Hypothesis 2).*

**Hypothesis 4.** *We restrict our analysis to safe programs (e.g. no division by 0, illegal indirect jump, segmentation fault, etc.). Under this hypothesis, concrete execution never gets stuck.*

**Definition 10** ($\approx_p^M$)**.** *We define a concretization relation $\approx_p^M$ between concrete and symbolic configurations, where $M$ is a model and $p \in \{l, r\}$ is a projection on the left or right side of a symbolic configuration. Intuitively, the relation $c \approx_p^M s$ is the concretization of the $p$-side of the symbolic state $s$ with the model $M$. Let $c \triangleq (l_1, r, m)$ and $s \triangleq (l_2, \rho, \widehat{\mu}, \pi)$. Formally $c \approx_p^M s$ holds iff $M \vDash \pi$, $l_1 = l_2$ and for all expression $e$, either the symbolic evaluation of $e$ gets stuck or we have*

$$(\rho, \widehat{\mu})\ e \vdash \widehat{\varphi} \ \wedge\ (M(\widehat{\varphi}_{|p}) = \texttt{bv} \iff c\ e \vdash \texttt{bv})$$

Notice that because both sides of an initial configuration $s_0$ are low-equivalent, the following proposition holds:

**Proposition 3.** *For all concrete configurations $c_0$ and $c_0'$ such that $c_0 \approx_l^M s_0 \wedge c_0' \approx_r^M s_0$, then $c_0 \simeq_l c_0'$.*

The following lemma expresses that when the symbolic evaluation is stuck on a state $s_k$, there exist concrete configurations derived from $s_k$ which produce distinct leakages.

**Lemma 1.** *Let $s_k$ be a symbolic configuration obtained after $k$ steps. If $s_k$ is stuck, then there exists a model $M$ such that for each concrete configurations $c_k \approx_l^M s_k$ and $c'_k \approx_r^M s_k$, the execution from $c_k$ and $c'_k$ produce distinct leakages.*

PROOF OVERVIEW: (Full proof in Appendix A.1.1) Proof by case analysis on the symbolic evaluation of $s_k$: for each symbolic step in which $s_k$ is stuck (i.e. a symbolic leakage predicate evaluates to $false$ with a model $M$), then $s_k$ can be concretized with the model $M$, producing states $c_k$ and $c'_k$ such that $c_k \xrightarrow[t]{} c_{k+1}$ and $c'_k \xrightarrow[t']{} c'_{k+1}$ and $t \neq t'$. This follows from the fact that the symbolic leakage model does not over-approximate the concrete leakage, i.e. each symbolic leak corresponds to a concrete leak. □

The following lemma expresses that when symbolic evaluation does not get stuck up to $k$, then for each pair of concrete executions following the same path up to $k$, there exists a corresponding symbolic execution.

**Lemma 2.** *Let $s_0$ be a symbolic initial configuration for a program $\mathbb{P}$ that does not get stuck up to $k$. For every concrete states $c_0$, $c_k$, $c'_0$, $c'_k$ and model $M$ such that $c_0 \approx_l^M s_0 \ \wedge \ c'_0 \approx_r^M s_0$, if $c_0 \xrightarrow[t]{}^k c_k$ and $c'_0 \xrightarrow[t']{}^k c'_k$ follow the same path, then there exists a symbolic configuration $s_k$ and a model $M'$ such that:*

$$s_0 \rightsquigarrow^k s_k \ \wedge \ c_k \approx_l^{M'} s_k \ \wedge \ c'_k \approx_r^{M'} s_k$$

PROOF OVERVIEW: (Full proof in Appendix A.1.2) Proof by induction on the number of steps $k$: for each concrete step $c_{k-1} \to c_k$ and $c'_{k-1} \to c'_k$, we show that, as long as they follow the same path, there is a symbolic step from $s_{k-1}$ to a state $s'_k$ that models $c_k$ and $c'_k$. This follows from the fact that our symbolic execution does not make under-approximations. □

#### 4.4.3.1 Correctness of RelSE

The following theorem claims the correctness of our symbolic execution, meaning that for each symbolic execution and model $M$ satisfying the path predicate, the concretization of the symbolic execution with $M$ corresponds to a valid concrete execution (no over-approximation).

**Theorem 1** (Correctness of RelSE)**.** *For every symbolic configurations $s_0$, $s_k$ such that $s_0 \rightsquigarrow^k s_k$ and for every concrete configurations $c_0$, $c_k$ and model $M$, such that $c_0 \approx_p^M s_0$ and $c_k \approx_p^M s_k$, there exists a concrete execution $c_0 \to^k c_k$.*

PROOF OVERVIEW: (Full proof in Appendix A.1.3) Proof by induction on the number of steps $k$: for each symbolic step $s_{k-1} \rightsquigarrow s_k$ and model $M_k$ such that $c_{k-1} \approx_p^{M_k} s_{k-1}$ and $c_k \approx_p^{M_k} s_k$, there exists a step $c_{k-1} \to c_k$ in concrete execution. For each rule, we show that there exists a unique step from $c_{k-1}$ to a state $c'_k$ (from Hypothesis 4 and Proposition 2), and, because there is no over-approximation in symbolic execution, $c'_k$ satisfies $c'_k \approx_p^{M_k} s_k$. □

#### 4.4.3.2 Correct bug-finding

The following theorem expresses that when the symbolic execution gets stuck, then the program is not constant-time.

**Theorem 2** (Bug-Finding for CT)**.** *Let $s_0$ be an initial symbolic configuration for a program $\mathbb{P}$. If symbolic evaluation gets stuck in a configuration $s_k$ then $\mathbb{P}$ is not constant-time at step $k$. Formally, if there is a symbolic evaluation $s_0 \rightsquigarrow^k s_k$ such that $s_k$ is stuck, then there exists a model $M$ and concrete configurations $c_0 \approx_l^M s_0$, $c'_0 \approx_r^M s_0$, $c_k \approx_l^M s_k$ and $c'_k \approx_r^M s_k$ such that,*

$$c_0 \simeq_l c'_0 \ \wedge \ c_0 \xrightarrow[t]{}^k c_k \xrightarrow[t_k]{} c_{k+1} \ \wedge \ c'_0 \xrightarrow[t']{}^k c'_k \xrightarrow[t'_k]{} c_{k+1} \wedge t_k \neq t'_k$$

*meaning that $\mathbb{P}$ is not constant-time at step $k$.*

PROOF: Let us consider symbolic configurations $s_0$ and $s_k$ such that $s_0 \rightsquigarrow^k s_k$ and $s_k$ is stuck. From Lemma 1, there is a model $M$ and concrete configurations $c_k$ and $c'_k$ such that $c_k \approx_l^M s_k$ and $c'_k \approx_r^M s_k$, and $c_k \xrightarrow[t_k]{} c_{k+1}$ and $c'_k \xrightarrow[t'_k]{} c'_{k+1}$ with $t_k \neq t'_k$. Additionally, let $c_0, c'_0$ be concrete configurations such that $c_0 \approx_l^M s_0$ and $c'_0 \approx_r^M s_0$. From Proposition 3, we have $c_0 \simeq_l c'_0$, and from Theorem 1, there are concrete executions $c_0 \xrightarrow[t]{}^k c_k$ and $c'_0 \xrightarrow[t']{}^k c'_k$. Therefore, we have $c_0 \xrightarrow[t]{}^k c_k \xrightarrow[t_k]{} c_{k+1}$ and $c'_0 \xrightarrow[t']{}^k c'_k \xrightarrow[t'_k]{} c'_{k+1}$ with $c_0 \simeq_l c'_0$ and $t_k \neq t'_k$, meaning that $\mathbb{P}$ is not constant-time at step $k$. □

#### 4.4.3.3 Relative completeness of RelSE

The following theorem claims the completeness of our symbolic execution relatively to an initial symbolic state. If the program is constant-time up to $k$, then for each pair of concrete executions up to $k$, there exists a corresponding symbolic execution (no under-approximation). Notice that our definition of completeness differs from standard definitions of completeness in SE [63]. Here, completeness up to $k$ only applies to programs that are constant-time up to $k$. This directly follows from the fact that our symbolic evaluation blocks on errors whereas concrete execution continues.

**Theorem 3** (Relative Completeness of RelSE)**.** *Let $\mathbb{P}$ be a program constant-time up to $k$ and $s_0$ be a symbolic initial configuration for $\mathbb{P}$. For every concrete states $c_0$, $c_k$, $c'_0$, $c'_k$, and model $M$ such that $c_0 \approx_l^M s_0 \ \wedge \ c'_0 \approx_r^M s_0$, if $c_0 \xrightarrow[t]{}^k c_k$ and $c'_0 \xrightarrow[t]{}^k c'_k$ then there exists a symbolic configuration $s_k$ and a model $M'$ such that:*

$$s_0 \rightsquigarrow^k s_k \ \wedge \ c_k \approx_l^{M'} s_k \ \wedge \ c'_k \approx_r^{M'} s_k$$

PROOF: First, note that from Theorem 2 and the hypothesis that $\mathbb{P}$ is constant-time up to $k$, we know that symbolic evaluation from $s_0$ does not get stuck up to $k$. Knowing this, we can apply Lemma 2 which directly entails Theorem 3. □

#### 4.4.3.4 Correct bounded-verification

Finally, we prove that if symbolic execution does not get stuck due to a satisfiable insecurity query, then the program is constant-time.

**Theorem 4** (Bounded-Verification for CT)**.** *Let $s_0$ be a symbolic initial configuration for a program $\mathbb{P}$. If the symbolic evaluation does not get stuck up to $k$, then $\mathbb{P}$ is constant-time up to $k$ w.r.t. $s_0$. Formally, if $s_0 \rightsquigarrow^k s_k$ then for all initial configurations $c_0$ and $c'_0$ and model $M$ such that $c_0 \approx_l^M s_0$, and $c'_0 \approx_r^M s_0$,*

$$c_0 \xrightarrow[t]{}^k c_k \ \wedge \ c'_0 \xrightarrow[t']{}^k c'_k \implies t = t'$$

*Additionally, if $s_0$ is fully symbolic and the execution does not get stuck for any $k$, then $\mathbb{P}$ is constant-time.*

PROOF OVERVIEW: (Full proof in Appendix A.1.4) Proof by induction on the number of steps: if the program is constant-time up to $k-1$ (induction hypothesis) then from Lemma 2 there is a symbolic execution for any configurations $c_{k-1}$ and $c'_{k-1}$. If these configurations produce distinct leakages, then symbolic execution is stuck at step $k-1$ which is a contradiction. This relies on the fact that the symbolic leakage model does not under-approximate the concrete leakage, i.e. each concrete leak is captured by a symbolic leak. □

## 4.5 Implementation

We implemented our relational symbolic execution, BINSEC/REL, on top of the binary-level analyzer BINSEC [90]. BINSEC/REL takes as input an x86 or ARM executable, a specification of high inputs and an initial memory configuration (possibly fully symbolic). It performs bounded exploration of the program under analysis (up to a user-given depth), and reports identified violations together with counterexemples (i.e. initial configurations leading to the vulnerabilities). In case no violation is reported, if the initial configuration is fully symbolic and the program has been explored exhaustively then the program is *proven* secure.

**Overall architecture.** The overall architecture is illustrated in Figure 4.8. The `Disasm` module loads the executable and lifts the machine code to the DBA intermediate representation [28]. Then, the analysis is performed by the `Rel` module on the DBA code. The `Formula` module is in charge of building and simplifying formulas, and sending the queries to the SMT solver. The queries are exported to the SMTLib [29] standard which permits to interface with many off-the-shelf SMT solvers.

**Rel plugin.** The `Rel` plugin represents approximately 3.8k lines of Ocaml. It is composed of a *relational symbolic exploration* module and an *insecurity analysis* module. The symbolic exploration module:

1. chooses a path to explore,
2. updates the symbolic state and the path predicate according to the current instruction,
3. ensure that it is satisfiable.

The insecurity analysis module builds insecurity queries and ensures that they are not satisfiable.

We explore the program in a depth-first search manner and we rely on the Boolector SMT solver [191], currently the best on theory QF_ABV [233, 107].

**Usability.** Binary-level semantic analyzers tend to be harder to use than their source-level counterparts as inputs are more difficult to specify (details in Appendix B.2) and results more difficult to interpret (details in Appendix B.3). In order to mitigate these points, we propose easy input specification (using dummy functions) when source code is available or when analyzing a library, and a vizualisation mechanism (based on IDA, which highlight coverage and violations). Appendix B provides more details on the challenges of implementing and using binary-level verification tools and the solutions we adopted to improve usability.

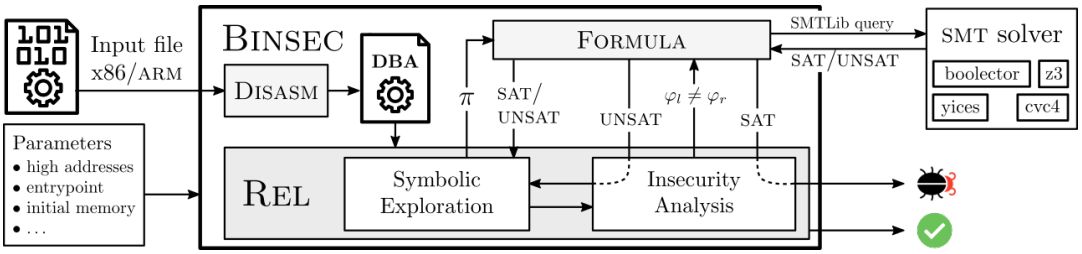



FIGURE 4.8 – BINSEC architecture with BINSEC/REL plugin.

## 4.6 Experimental evaluation

**Section overview**

This section evaluates our technique, binary-level RelSE, and tool, BINSEC/REL, on cryptographic implementations, answering several research questions (cf. Section 4.6.1). These questions address the effectiveness of our tool for bug-finding and bounded-verification (cf. Section 4.6.2); and the comparison of our technique against standard approaches (cf. Section 4.6.3).

### 4.6.1 Research questions and methodology

**Research questions.** We investigate the following research questions:

**RQ1. Effectiveness.** Is BINSEC/REL able to perform constant-time analysis on real cryptographic binaries, for both bug finding and bounded-verification?

**RQ2. Genericity.** Is BINSEC/REL generic enough to encompass several architectures and compilers?

**RQ3. Comparison vs. standard approaches.** How does BINSEC/REL scale compared to traditional approaches based on self-composition and RelSE?

**RQ4. Impact of simplifications.** What are the respective impacts of our different simplifications?

**RQ5. Comparison vs. SE.** What is the overhead of BINSEC/REL compared to standard SE, and can our simplifications be useful for standard SE?

To answer **RQ1**, we perform bug-finding and bounded-verification on a wide range of cryptographic primitives (cf. Sections 4.6.2.1 and 4.6.2.2). For **RQ2**, we extend and automate a prior study on constant-time preservation by compiler in Section 4.6.2.3, encompassing several compilers, compilation options, for both x86 and ARM architecture. To answer **RQ3**, we compare BINSEC/REL with self-composition and standard RelSE in Section 4.6.3.1. For **RQ4**, we perform an ablation study in Section 4.6.3.2 to measure the impact of individual optimizations implemented in BINSEC/REL. Finally, for **RQ5**, we compare BINSEC/REL with standard SE and evaluate the performance of our on-the-fly read-over-write against standard read-over-write implemented as a formula pre-processing (cf.Section 4.6.3.3).

**Metrics.** Throughout this section we use the following notations:

— Inst denotes the number of static instructions of a program,

— $I_{unr}$ is the number of unrolled instructions explored by the analysis,

— $I_{unr}/s$ is the number of unrolled instructions explored per second,

— $Q_{expl}$ is the number of exploration queries sent to the solver,

— $Q_{insec}$ is the number of insecurity queries sent to the solver,

— $Q_{tot}$ is the total number of queries sent to the solver,

— Time is the execution time given in seconds,

— 🐞 is the number of bugs (vulnerable instructions) found.

— Status is set to ✓ for secure (exhaustive exploration), ✗ for insecure, or ⧗ for timeout (set to 1 hour).

These metrics give a good overview of the efficiency ($I_{unr}$/s, Time, $Q_{tot}$, ⧗), effectiveness (🐞, ✓, ✗), and internal details ($Q_{expl}$, $Q_{insec}$) of the analysis.

**Setup.** Experiments were performed on a laptop with an Intel(R) Xeon(R) CPU E3-1505M v6 @ 3.00GHz processor and 32GB of RAM. Similarly to related work (e.g. [99]), `esp` is initialized to a concrete value, we start the analysis from the beginning of the `main` function, we statically allocate data structures and the length of keys and buffers is fixed.

**Benchmark.** We perform our experimental evaluation on a wide set of cryptographic primitives and utility functions. Table 4.2 details for each program, the type of operation performed, and the length of the secret key (Key) and message (Msg) when applicable (in bytes). Programs are compiled for a x86-32 bit architecture with their default compiler setup.

Overall, our study encompasses 338 representative binary codes for a total of 70k machine instructions and 22M unrolled instructions (i.e., instructions explored by Binsec/Rel).

### 4.6.2 Effectiveness of Binsec/Rel (RQ1-RQ2)

To assess the effectiveness of Binsec/Rel, we carry out three experiments:

1. Bounded-verification of secure cryptographic primitives previously verified at source- or LLVM-level [45, 10, 272] in Section 4.6.2.1;
2. Automatic replay of known bug studies [231, 10, 6] in Section 4.6.2.2;
3. Automatic study of constant-time preservation by compilers extending prior work [231] (Section 4.6.2.3).

#### 4.6.2.1 Bounded-verification (RQ1)

Using Binsec/Rel, we analyze a large range of *secure* cryptographic primitives—i.e. the 296 secure binary codes described in Table 4.2, for a total of 64k instructions—comprising:

— Several basic constant-time utility functions such as selection functions [231], sort functions [141] and utility functions from HACL* [272] and OpenSSL [192];

— A set of representative constant-time cryptographic primitives already studied in the literature on source code [45] or LLVM [10], including implementations of TEA [256], Curve25519-donna [40], `aes` and `des` encryption functions taken from BearSSL [207], cryptographic primitives from libsodium [41], and the constant-time padding remove function `tls-cbc-remove-padding` from OpenSSL [10];

— A set of functions from the HACL* library [272].

| Description and nb. of binaries† | | Status | ≈Inst | Type | Key | Msg |
|---|---|---|---|---|---|---|
| utility | ct-select [231] (×40) | 29 ✓ & 21 ✗ | 1750 | Utility functions | - | - |
| | ct-sort [141] (×30) | 12 ✓ & 18 ✗ | 6000 | | | |
| | Hacl* [272] (×110) | 110 ✓ | 3850 | | | |
| | OpenSSL [192] (×130) | 130 ✓ | 4550 | | | |
| tea [255] | decrypt `-O0` | ✓ | 290 | Block cipher | 16 | 8 |
| | decrypt `-O3` | ✓ | 250 | | | |
| donna [167] | `-O0` | ✓ | 7083 | Elliptic curve | 32 | - |
| | `-O3` | ✓ | 4643 | | | |
| libsodium [41] | salsa20 | ✓ | 1627 | Stream cipher | 32 | 256 |
| | chacha20 | ✓ | 2717 | Stream cipher | 32 | 256 |
| | sha256 | ✓ | 4879 | Secure hash | - | 256 |
| | sha512 | ✓ | 16312 | Secure hash | - | 256 |
| Hacl* [272] | chacha20 | ✓ | 1221 | Stream cipher | 32 | 256 |
| | sha256 | ✓ | 1279 | Secure hash | - | 256 |
| | sha512 | ✓ | 2013 | Secure hash | - | 256 |
| | curve25519 | ✓ | 8522 | Elliptic curve | 32 | - |
| BearSSL [207] | aes-ct-cbcenc‡ | ✓ | 357 | Block cipher | 240 | 32 |
| | aes-big-cbcenc‡ | ✗ | 375 | Block cipher | 240 | 32 |
| | des-ct-cbcenc‡ | ✓ | 682 | Block cipher | 384 | 16 |
| | des-tab-cbcenc‡ | ✗ | 365 | Block cipher | 384 | 16 |
| OpenSSL [9] | tls-rem-pad-patch | ✓ | 424 | Remove padding | - | 63 |
| | tls-rem-pad-lucky13 | ✗ | 950 | Remove padding | - | 63 |
| **Total** | 338 binaries | 296 ✓ & 42 ✗ | 70k | | | |

TABLE 4.2 – Cryptographic primitives and size of symbolic input used for our experimental evaluation where ✓ indicates secure programs and ✗ indicates insecure programs. † A line in which the number of binaries is not indicated corresponds to 1 binary. ‡ Bound set to 2 rounds.

Results are reported in Table 4.3. For each program, BINSEC/REL is able to perform an exhaustive exploration without finding any violations of constant-time in less than 20 minutes. Note that exhaustive exploration is possible because in cryptographic programs, bounding the input size bounds loops. These results show that BINSEC/REL can perform bounded-verification of real-world cryptographic implementations at binary-level in a reasonable time, which was impractical with previous approaches based on self-composition or standard RelSE (see Section 4.6.3).

*This is the first automatic constant-time-analysis of these cryptographic libraries at binary-level.*

#### 4.6.2.2 Bug-finding (RQ1)

Using BINSEC/REL, we automatically replay known bug studies from the literature [231, 141, 6]—i.e. the 42 insecure binary codes described in Table 4.2, for a total of 6k instructions—comprising:

1. binaries compiled from constant-time sources of a selection function [231] and sort functions [141],

| Description and nb. of binaries† | | ≈Inst | $I_{unr}$ | Time | Status |
|---|---|---|---|---|---|
| utility | ct-select (×29) | 1015 | 1507 | .2 | 29 × ✓ |
| | ct-sort (×12) | 2400 | 1782 | .2 | 12 × ✓ |
| | Hacl* (×110) | 3850 | 90953 | 7.6 | 110 × ✓ |
| | OpenSSL (×130) | 4550 | 5113 | .9 | 130 × ✓ |
| tea | decrypt `-O0` | 290 | 953 | .1 | ✓ |
| | decrypt `-O3` | 250 | 804 | .1 | ✓ |
| donna | `-O0` | 7083 | 10.2M | 1008.5 | ✓ |
| | `-O3` | 4643 | 2.7M | 347.1 | ✓ |
| libsodium | salsa20 | 1627 | 38.0k | 3.5 | ✓ |
| | chacha20 | 2717 | 12.3k | 1.5 | ✓ |
| | sha256 | 4879 | 48.4k | 4.5 | ✓ |
| | sha512 | 16312 | 92.0k | 8.1 | ✓ |
| Hacl* | chacha20 | 1221 | 6.7k | 4.3 | ✓ |
| | sha256 | 1279 | 21.0k | 2.7 | ✓ |
| | sha512 | 2013 | 47.5k | 5.2 | ✓ |
| | curve25519 | 8522 | 9.4M | 927.8 | ✓ |
| BearSSL | aes-ct-cbcenc | 357 | 3.5k | .5 | ✓ |
| | des-ct-cbcenc | 682 | 19.9k | 12.1 | ✓ |
| OpenSSL | tls-remove-padding-patch | 424 | 35.7k | 438.0 | ✓ |
| **Total** | 296 binaries | 64114 | 22.8M | 2772.7 | 296 × ✓ |

TABLE 4.3 – Performance of BINSEC/REL: bounded-verification for constant-time on cryptographic implementations. † A line in which the number of binaries is not indicated corresponds to 1 binary.

2. non-constant-time versions of `aes` and `des` from BearSSL [207] (`aes_big` and `des_tab`),
3. non-constant-time version of OpenSSL `tls-cbc-remove-padding`[6] responsible for the famous Lucky13 attack [6].

Results are reported in Table 4.4 with *fault-packing disabled* to report vulnerabilities at the instruction level. All bugs have been found within a 1 hour timeout. Interestingly, we found three *unexpected binary-level vulnerabilities* (from secure source codes) that slipped through previous analysis:

— function `ct_select_v1` was deemed secured through binary-level manual inspection [231], still we confirm that any version of `clang` with `-O3` introduces a secret-dependent conditional jump which violates constant-time;

— functions `ct_sort` and `ct_sort_mult`, verified by ct-verif [10] (LLVM bitcode compiled with `clang`), are vulnerable when compiled with `gcc -O0` or `clang -O3 -m32 -march=i386`.

More details on these vulnerabilities are provided in Section 4.6.2.3. Finally, we describe the application of BINSEC/REL to the Lucky13 attack in Appendix C.2.1.

6. https://github.com/openssl/openssl/blob/OpenSSL_1_0_1/ssl/d1_enc.c

| Description and nb. of binaries† | | ≈Inst | $I_{unr}$ | Time | CT | Status | 🐞 |
|---|---|---|---|---|---|---|---|
| utility | ct-select (×21) | 735 | 767 | 0.4 | Y | 21 × ✗ (1 new) | 21 |
| | ct-sort (×18) | 3600 | 7513 | 13.6 | Y | 18 × ✗ (2 new) | 44 |
| BearSSL | aes-big-cbcenc | 375 | 876 | 1651.8 | N | ✗ | 32 |
| | des-tab-cbcenc | 365 | 5187 | 4.4 | N | ✗ | 8 |
| OpenSSL | tls-rem-pad-lucky13 | 950 | 7866 | 700.3 | N | ✗ | 6 |
| **Total** | 42 binaries | 6025 | 22209 | 2370.5 | - | 42 × ✗ | 111 |

TABLE 4.4 – Performance of BINSEC/REL: bug-finding for constant-time on cryptographic implementations. CT indicates if the program is constant-time at source level (Y) or not (N). † A line in which the number of binaries is not indicated corresponds to 1 binary.

#### 4.6.2.3 Preservation of constant-time by compilers (RQ1-RQ2)

Simon *et al.* [231] *manually* analyze whether `clang` optimizations break the constant-time property, for 5 different versions of a selection function. We reproduce their analysis in an *automatic* manner and extent it for a total of 408 binaries (192 in the initial study), adding:

— 29 new functions, including utility functions from OpenSSL [192] and HACL* [272], and TEA [256] encryption primitive;

— a newer version of `clang` (`clang-7.1`);

— the `gcc` compiler versions `5.4` and `8.3`;

— the ARM architecture with compiler `arm-linux-gnueabi-gcc`.

Results are presented in Table 4.5.

| | cl-3.0 | | cl-3.9 | | **cl-7.1** | | **gcc-5.4** | | **gcc-8.3** | | **arm-gcc** | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | `-O0` | `-O3` | `-O0` | `-O3` | `-O0` | `-O3` | `-O0` | `-O3` | `-O0` | `-O3` | `-O0` | `-O3` |
| ct_select_v1 | ✓ | (✗) | ✓ | (✗) | ✓ | ✗ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| ct_select_v2 | ✓ | ✗ | ✓ | ✗ | ✓ | ✗ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| ct_select_v3 | ✓ | ✓ | ✓ | ✗ | ✓ | ✗ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| ct_select_v4 | ✓ | ✗ | ✓ | ✗ | ✓ | ✗ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| select_naive (insecure) | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✓ |
| **ct_sort** | ✓ | ✗ | ✓ | ✗ | ✓ | ✓ | ✗ | ✓ | ✗ | ✓ | ✗ | ✓ |
| **ct_sort_mult** | ✓ | ✗ | ✓ | ✗ | ✓ | ✓ | ✗ | ✓ | ✗ | ✓ | ✗ | ✓ |
| **sort_naive** (insecure) | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✓ |
| **hacl_utility** (×11) | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| **openssl_utility** (×13) | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| **tea_encrypt** | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| **tea_decrypt** | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |

TABLE 4.5 – Constant-time analysis of functions compiled with `gcc` or `clang` (`cl`) and optimization levels `O0` or `O3`. ✓ indicates that a program is secure and ✗ that it is insecure. **Bold programs** and **compilers** are extensions of [231] and (✗) indicates a different result than [231].

We confirm the main conclusion of Simon *et al.* [231] that `clang` is more likely to optimize away constant-time protections as the optimization level increases. However, *contrary to their work*, our experiments show that newer versions of `clang` are not necessarily more likely than older ones to break constant-time—e.g. `ct_sort` is compiled to a non-constant-time code with `clang-3.9` but not with `clang-7.1`.

Surprisingly, in contrast with `clang`, `gcc` optimizations tend to remove branches and thus, are less likely to introduce vulnerabilities in constant-time code. Especially, `gcc` *for ARM produces secure binaries from the insecure source codes*[7] `sort_naive` and `select_naive`.

Although [231] reports that the `ct_select_v1` function is secure in all their settings, we find the opposite. Manual inspection confirms that `clang` with `-O3` introduces a secret-dependent conditional jump violating constant-time.

Finally, as discussed in Section 4.6.2.2, we found that the `ct_sort` and `ct_sort_mult` functions, taken from the benchmark of the `ct-verif` [10] tool, can be compiled to insecure binaries. Those vulnerabilities are out of reach of `ct-verif` because it targets LLVM code compiled with `clang`, whereas the vulnerabilities are either introduced by `gcc` or by *backend passes* of `clang`—we did confirm that `ct-verif` with the setting `-clang-options="-O3 -m32 -march=i386"` does not report the vulnerability.

**Conclusion (RQ1-RQ2).** We perform an extensive analysis over 338 binary codes of cryptographic primitives and utility functions studied in the literature [45, 272, 10], which demonstrates that BINSEC/REL scales to realistic applications for both bug-finding and bounded-verification (RQ1). We apply BINSEC/REL on programs compiled with multiple versions and options of `clang` and `gcc`, over x86 and ARM, which demonstrates that the technology is generic (RQ2). We also get the following interesting side results:

— We proved constant-time-secure 296 binaries of interest;

— We found 3 new vulnerabilities that slipped through prior analysis—manual on binary code [231] or automated on LLVM [10];

— We extend and automate a previous study on effects of compilers on constant-time [231]—from 192 configurations to 408;

— We found that `gcc` optimizations tend to help enforcing constant-time—on ARM, `gcc` even sometimes produces secure binaries from insecure sources.

### 4.6.3 Comparison with standard techniques (RQ3-RQ4-RQ5)

We compare BINSEC/REL with standard techniques based on self-composition and relational symbolic execution (*RelSE*) in Section 4.6.3.1, then we analyze the performance of our different simplifications in Section 4.6.3.2, and finally we investigate the overhead of BINSEC/REL compared to standard symbolic execution, and whether the *FlyRow* simplifications is useful for symbolic execution (Section 4.6.3.3).

Experiments are performed on the programs introduced in Section 4.6.2 for bug-finding and bounded-verification, including secure and insecure examples for a total of 338 binaries and 70k instructions.

7. The compiler takes advantage of the fact that ARM has many conditional instructions to remove conditional jumps.

#### 4.6.3.1 Comparison with self-composition and RelSE (RQ3)

We evaluate BINSEC/REL against self-composition (*SC*) and *RelSE*. Since no implementation of these methods fits our particular use-cases, we implement them directly in BINSEC. *RelSE* is obtained by disabling BINSEC/REL optimizations (Section 4.4.2), whereas *SC* is implemented on top of *RelSE* by duplicating low inputs instead of sharing them and adding the adequate preconditions. Results are given in Table 4.6.

| | $I_{unr}$ | $I_{unr}/s$ | $Q_{tot}$ | $Q_{expl}$ | $Q_{insec}$ | Time | ⧗ | ✓ | ✗ |
|---|---|---|---|---|---|---|---|---|---|
| *SC* | 248k | 3.9 | 158k | 14k | 143k | 64296 | 16 | 280 | 42 |
| *RelSE* | 349k | 6.2 | 90k | 17k | 73k | 56428 | 13 | 283 | 42 |
| BINSEC/REL | 22.8M | 6238 | 3700 | 2408 | 1292 | 3657 | 0 | 296 | 42 |

TABLE 4.6 – BINSEC/REL vs. self-composition (*SC*) and *RelSE*

While *RelSE* performs slightly better than *SC* (1.6× speedup in terms of instructions per second) thanks to a noticeable reduction of the number of queries (approximately 50%), both techniques are not efficient enough on binary code: *RelSE* times out in 13 cases and achieves an analysis speed of only 6.2 instructions per second whereas *SC* times out in 16 cases and achieves an analysis speed of only 3.9 instruction per second. BINSEC/REL completely outperforms both previous approaches:

— The optimizations implemented in BINSEC/REL drastically reduce the number of queries sent to the solver (57× less insecurity queries than *RelSE*);

— BINSEC/REL reports no timeout, is 1000× faster than *RelSE* and 1600× faster than *SC* in terms of instructions explored per second;

— BINSEC/REL can perform bounded-verification of large programs (e.g. `donna`, `des_ct`, `chacha20`, etc.) that were out of reach of standard methods.

#### 4.6.3.2 Performance of simplifications (RQ4)

We evaluate the performance of our individual optimizations: on-the-fly read-over-write (*FlyRow*), untainting (*Unt*) and fault-packing (*FP*). Results are reported in Table 4.7:

— *FlyRow* is the major source of improvement in BINSEC/REL, drastically reducing the number of queries sent to the solver and allowing a 718× speedup compared to *RelSE* in terms of instructions per second;

— Untainting and fault-packing do have a positive impact on *RelSE*—untainting alone reduces the number of queries by almost 50%, the two optimizations together yield a 2× speedup;

— Yet, their impact is more modest once *FlyRow* is activated: untainting leads to a very slight slowdown, while fault-packing achieves a 1.4× speedup.

It is worth noting that *FP* can be interesting on some particular programs, when the precision of the bug report is not the priority. Consider for instance the non-constant-time version of `aes` in BearSSL (i.e. `aes_big`). On this program, BINSEC/REL without *FP* reports 32 vulnerable instructions in 1580 seconds, whereas BINSEC/REL with *FP* reports 2 vulnerable *basic blocks*, covering the 32 vulnerable instructions, in only 146 seconds (almost 11× faster).

| Version | $I_{unr}$ | $I_{unr}$/s | $Q_{tot}$ | $Q_{expl}$ | $Q_{insec}$ | Time | ⧗ | ✓ | ✗ |
|---|---|---|---|---|---|---|---|---|---|
| **Standard RelSE with *Unt* and *FP*** | | | | | | | | | |
| *RelSE* | 349k | 6.2 | 90148 | 17428 | 72720 | 56429 | 13 | 283 | 42 |
| + *Unt* | 414k | 9.9 | 48648 | 20601 | 28047 | 41852 | 7 | 289 | 42 |
| + *FP* | 437k | 12.7 | 35100 | 21834 | 13266 | 34471 | 7 | 289 | 42 |
| **BINSEC/REL (*RelSE* + *FlyRow* + *Unt* + *FP*)** | | | | | | | | | |
| *RelSE* + *FlyRow* | 22.8M | 4450 | 3738 | 2408 | 1330 | 5127 | 0 | 296 | 42 |
| + *Unt* | 22.8M | 4429 | 3738 | 2408 | 1330 | 5151 | 0 | 296 | 42 |
| + *FP* | 22.8M | 6238 | 3700 | 2408 | 1292 | 3658 | 0 | 296 | 42 |

TABLE 4.7 – Performance of BINSEC/REL simplifications.

#### 4.6.3.3 Comparison against standard symbolic-execution (RQ5).

We investigate the overhead of BINSEC/REL compared to standard symbolic execution (*SE*); evaluate whether on-the-fly read-over-write (*FlyRow*) can improve performance of *SE*; and also compare *FlyRow* to a recent implementation of read-over-write [107] (*PostRow*), implemented posterior to symbolic-execution as a formula preprocessing. Standard symbolic-execution is directly implemented in the REL module and models a single execution of the program with exploration queries but *without insecurity queries*.

Results are presented in Table 4.8:

— BINSEC/REL, compared to our best setting for symbolic execution (*SE+FlyRow*), only has an overhead of 2× in terms of instructions per second. Hence constant-time comes with an acceptable overhead on top of standard symbolic execution. This is consistent with the fact that our simplifications discard most insecurity queries, letting only the exploration queries which are also part of symbolic-execution;

— For RelSE, *FlyRow* completely outperforms *PostRow*. First, *PostRow* is not designed for relational verification and duplicates the memory. Second, *PostRow* simplifications are not propagated along the execution and must be recomputed for every query, producing a significant simplification overhead. On the contrary, *FlyRow* models a single memory containing relational values and propagates along symbolic execution;

— *FlyRow* also improves the performance of standard symbolic execution by a factor 643 in our experiments, performing much better than *PostRow* (430× faster).

**Conclusion (RQ3-RQ4-RQ5).** BINSEC/REL performs significantly better than previous approaches to relational symbolic execution (1000× speedup vs. *RelSE*). The main source of improvement is the on-the-fly read-over-write simplification (*FlyRow*), which yields a 718× speedup vs. *RelSE* and sends 57× less insecurity queries to the solver.

Note that, in our context, *FlyRow* outperforms state-of-the-art binary-level simplifications, as they are not designed to efficiently cope with relational properties and introduce a significant simplification-overhead at every query.

Fault-packing and untainting, while effective over *RelSE*, have a much slighter impact once *FlyRow* is activated; fault-packing can still be useful on insecure programs.

| Version | $I_{unr}$ | $I_{unr}/s$ | $Q_{tot}$ | Time | ⧗ |
|---|---|---|---|---|---|
| *SE* | 522k | 19.5 | 24444 | 26814 | 6 |
| *SE+PostRow* [107] | 628k | 29.2 | 29389 | 21475 | 4 |
| *SE+FlyRow* | 22.8M | 12531 | 534 | 1817 | 0 |
| *RelSE* | 349k | 6.2 | 90148 | 56428 | 13 |
| *RelSE+PostRow* | 317k | 5.3 | 65834 | 60295 | 15 |
| *RelSE+FlyRow* | 22.8M | 4450 | 3738 | 5127 | 0 |
| BINSEC/REL | 22.8M | 6238 | 3700 | 3657 | 0 |

TABLE 4.8 – Performance of RelSE compared to standard symbolic execution with/without binary level simplifications.

Finally, in our experiments, *FlyRow* significantly improves performance of standard symbolic-execution (643× speedup), performing better than read-over-write implemented posterior to symbolic-execution as a formula pre-processing (430× faster than *PostRow*).

## 4.7 Discussion

**Implementation limitations.** The implementation of BINSEC/REL has the same limitations as BINSEC, discussed in Section 2.3.3.2:

— It is limited to x86-32, ARMv7 and RiscV instruction set;

— It does not support dynamic libraries (executable must be statically linked or stubs must be provided for external function calls);

— It does not implement predefined stubs for system calls;

— It does not support dynamic allocation;

— It does not support floating point instructions.

These limitations are commonly found in research prototypes and are orthogonal to the core contribution of this paper. Moreover, the prototype is already efficient on real-world case studies.

**Threats to validity in experimental evaluation.** We assessed the effectiveness of our tool on several known secure and insecure real-world cryptographic binaries, many of them taken from prior studies. All results have been crosschecked with the expected output, and manually reviewed in case of deviation.

Our prototype is implemented as part of BINSEC [90], whose efficiency and robustness have been demonstrated in prior large scale studies on both adversarial code and managed code [27, 211, 106, 89]. The lifting to an intermediate representation has been positively evaluated in an external study [149] and the symbolic engine features aggressive formula optimizations [107]. All our experiments use the same search heuristics (depth-first) and, for bounded-verification, smarter heuristics do not change the performance. Regarding the solver, we tried z3 [92] and confirmed the better performance of Boolector [8].

Finally, we compare our tool to our own versions of self-composition (*SC*) and *RelSE*, primarily because none of the existing tools can be easily adapted for our setting, and also because it allows us to compare very close implementations.

8. Since 2020, the best solver in this category is `bitwizla` [190] and support for this solver has been recently added to BINSEC

## 4.8 Related work

**Section overview**

Related work on information-flow analysis and timing attacks has already been discussed in Section 3.1 and Section 3.2. This section discusses existing SE-based tools for information analysis and automatic analyzers for constant-time.

SE-based tools for information flow analysis are summarized in Table 4.9. Automatic analyzers for constant-time are summarized in Table 4.10 (partly taken from [10]). In the tables, we report whether the tools can prove that a program is secure (P), perform bounded-verification (BV), do correct bug-finding with no false alarms (BF).

| Tool | Target | NI | Technique | P | BV | BF | ≈XP | $I_u$/s |
|---|---|---|---|---|---|---|---|---|
| RelSym [105] | imp-for | ✓ | RelSE | ✓ | ✓ | ✓ | 10 LoC | NA |
| IF-exploit [98] | Java | ✓ | SC | ✓ | ✓ | ✓ | 20 LoC | NA |
| Type-SC-SE [182] | C | ✓ | type-based SC+DSE | ✗ | ✗ | ✓ | 20 LoC | NA |
| Casym [49] | LLVM | ✓ | SC+over-approx | ✓ | ✓ | ✗ | 200 LoC | NA |
| ENCoVer [25] | Java bytecode | ✓ | epistemic logic+DSE | ✗ | ✓ | ✓ | 33k $I_u$ | 186 |
| IF-low-level [24] | **binary** | ✓ | SC+invariants | ✓ | ✓ | ✗ | 250 $I_s$ | NA |
| IF-firmware [238] | **binary** | ✗ | SC+concretization | ✗ | ✗ | ✓ | 500k $I_u$ | 260 |
| CacheD [254] | **binary** | ✗ | DSE+SC | ✗ | ✗ | ✓ | 31M $I_u$ | 2010 |
| Binsec/Rel | **binary** | ✗ | RelSE+simplifications | ✗ | ✓ | ✓ | 10M $I_u$ | 6238 |

Table 4.9 – Comparison of SE-based tools for information flow analysis. NI indicates whether the tool handles general non-interference (diverging paths) or not. In the column "Technique", DSE stands for dynamic symbolic execution, and SC for self-composition. ≈XP indicates the maximum size of the use cases where LoC stands for Lines of Code, $I_s$ for static instructions, and $I_u$ for unrolled instructions. Finally, NA indicates Non-Applicable.

**Self-composition and symbolic execution.** Symbolic execution has first been combined with self-composition by Milushev *et al.* [182]. They use type-directed self-composition and dynamic symbolic execution based on KLEE [58] to find bugs of *noninterference* in C programs. However they do not address scalability and their experiments are limited to toy programs. The main issues here are the quadratic explosion of the search space (due to the necessity of considering diverging paths) and the complexity of the underlying formulas. Later works [98, 24] suffer from the same problems.

*Instead of considering the general case of noninterference, we focus on constant-time, and we show that it remains tractable for symbolic execution with adequate optimizations.*

**Relational symbolic execution.** *Shadow symbolic execution* [62, 196, 159] aims at efficiently testing evolving software by focusing on the new behaviors introduced by a patch. It introduces the idea of *sharing formulas* across two executions in the same symbolic execution instance. The term *relational symbolic execution* has been coined more recently [105] and applies similar concepts to the analysis of 2-hypersafety

properties. However, this work is limited to a simple toy imperative language and do not address scalability.

*We maximize sharing between pairs of executions, as ShadowSE does, but we also develop specific optimizations tailored to the case of constant-time analysis at binary-level. Experiments show that our optimizations are crucial in this context.*

**Scaling symbolic execution for information flow analysis.** Only three previous works achieve scalability when applying symbolic execution for information flow analysis, yet at the cost of either precision or soundness. Wang et al. [254] and Subramanyan *et al.* [238] sacrifice soundness for scalability (no bounded-verification). The former performs symbolic execution on fully concrete traces and only symbolizes secrets. The latter concretizes memory accesses. In both cases, they may miss feasible paths as well as vulnerabilities. Brotzman *et al.* [49] take the opposite side and sacrifice precision for scalability (no bug-finding). Their analysis scales by over-approximating loops and resetting the symbolic state at chosen code locations.

*We adopt a different approach and scale by heavy formula optimizations, allowing us to keep both correct bug-finding and correct bounded-verification.* Interestingly, our method is faster than these approximated ones. Moreover, our technique is compatible with the previous approximations for extra-scaling.

**Other methods for constant-time analysis.** *Static approaches* based on sound static analysis [4, 183, 30, 21, 45, 164, 99, 100, 10, 216] give formal guarantees that a program is free from timing-side-channels but they cannot find bugs when a program is rejected. Some works also propose program transformations to make a program secure [163, 4, 183, 70, 260, 49] but they target higher-level code and cannot detect vulnerabilities in existing binary codes. *Dynamic approaches* for constant-time are precise (they find real violations) but limited to a subset of the execution traces, hence they are not complete. These techniques include statistical analysis [214], dynamic binary instrumentation [166, 257], and dynamic symbolic execution (DSE) [69], or fuzzing [138]..

| Tool | Target | Analysis | Technique | P | BV | BF |
|---|---|---|---|---|---|---|
| ct-ai [45] | C | static | abstract-interpretation | ✓ | ✓ | ✗ |
| FlowTracker [216] | LLVM | static | type-system | ✓ | ✓ | ✗ |
| ct-verif [10] | LLVM | static | logical, product-programs | ✓ | ✓ | ✗* |
| Casym‡ [49] | LLVM | static | SE + over-approx. | ✓ | ✓ | ✗ |
| VirtualCert† [30] | x86 | static | type-system | ✓ | ✓ | ✗ |
| ctgrind [166] | **binary** | dynamic | Valgrind [186] | ✗ | ✗ | ✗ |
| CacheAudit‡ [100] | **binary** | static | abstract-interpretation | ✓ | ✓ | ✗ |
| CacheD‡ [254] | **binary** | dynamic | DSE | ✗ | ✗ | ✓ |
| BINSEC/REL | **binary** | static | RelSE + simplifications | ✗ | ✓ | ✓ |

TABLE 4.10 – Automatic analysis tools for properties ensuring the absence of timing side-channel (see [10]) where DSE stands for dynamic symbolic execution. *ct-verif can be incomplete because of invariant inference. †As part of CompCert, cannot be used on arbitrary executables. ‡Also implements a cache model.

## 4.9 Conclusion

We tackle the problem of designing an automatic and efficient binary-level analyzer for *constant-time* properties, enabling both bug-finding and bounded-verification on real-world cryptographic implementations. Our approach is based on *relational symbolic execution* together with original *dedicated optimizations* reducing the overhead of relational reasoning and allowing for a significant speedup. Our prototype, Binsec/Rel, is shown to be highly efficient compared to alternative approaches. We used it to perform extensive binary-level constant-time analysis for a wide range of cryptographic implementations, and to automate prior manual studies on the preservation of constant-time by compilers. While conducting this study, we found three constant-time vulnerabilities that slipped through previous manual and automated analyses, and we discovered that `gcc -O0` and backend passes of `clang` introduce violations of constant-time out of reach of state-of-the-art constant-time verification tools at LLVM or source level.

# Chapter 5

# Generalization of BINSEC/REL and Application to Secret-Erasure

**Chapter overview**

In this chapter, we generalize binary-level symbolic execution—introduced in Chapter 4 for constant-time—to a subset of information flow properties. This subset encompasses *constant-time* and *secret-erasure*, two properties that are crucial in cryptographic implementation but are generally not preserved by compilers. We adapt BINSEC/REL to verify secret-erasure, and propose an easily extensible framework to verify, in various setups, that compilers do not introduce violations of secret-erasure. Using this framework, we automate and extend a prior manual study on preservation secret-erasure by compilers. Interestingly, our analysis highlights incorrect usages of volatile data pointers for secret erasure and shows that enforcement mechanisms based on *volatile function pointers* can introduce additional register spilling which might break secret-erasure.

## 5.1 Introduction

Secret-erasure [74] (a.k.a. data scrubbing or safe erasure) requires to clear secret data (e.g. secret keys) from the memory after the execution of a critical function, for instance by zeroing out the corresponding memory. It ensures that secret data do not persist in memory longer than necessary, protecting them against subsequent memory disclosure vulnerabilities.

**Problem.** Scrubbing operations used for secret-erasure have no effect on the result of the program and can therefore be optimized away by the dead-store-elimination pass of the compiler [269, 43, 101], as detailed in CWE-14 [83]. Moreover, these scrubbing operations do not erase secrets that have been copied on the stack by compilers from register spilling—when there are not enough registers to hold all the variables and some of them are moved to the stack.

Unfortunately, there is currently no sound automatic analyzer for secret-erasure. Existing approaches rely on dynamic tainting [227, 231] or manual binary-code analysis [269], thus they can miss vulnerabilities in unexplored parts of the code. While there has been some work on security preserving compilers [43, 231], they are not always applicable and are ineffective for detecting errors in existing binaries.

**Proposal.** We extend binary-level RelSE defined in Section 4.4 to a subset of information flow properties relating traces following the same path—which includes interesting security policies such as constant-time. We propose a novel formalization of secret-erasure expressible in this framework. We make BINSEC/REL modular in the property to verify and extend it with the secret-erasure property. BINSEC/REL is the first efficient binary-level automatic tool for bug-finding and bounded-verification of constant-time and secret-erasure at binary-level.

**Contributions.** The contributions detailed in this chapter are the following:

— The leakage model considered in Section 4.4 is restricted to constant-time whereas this work encompasses a more general subset of information flow properties. In particular, we define a new leakage model and property to capture the notion of *secret-erasure* (cf. Sections 5.3.1 and 5.3.2). We also discuss the adaptation of the theorems to other information-flow properties in Section 5.4.3;

— We adapt the BINSEC/REL tool to be modular in the property to check and extend it to verify the *secret-erasure* property (Section 5.5);

— Finally, we build the first framework to automatically check the preservation of secret-erasure by compilers. We use it to analyze 17 scrubbing functions—including countermeasures manually analyzed in a prior study [269], compiled with 10 compilers at 4 different optimization levels, for a total of 680 binaries (cf. Section 5.6). Our analysis:

1. Confirms that secret-erasure mechanisms based on dedicated secure functions (i.e. `explicit_bzero`, `memset_s`), memory barriers, and weak symbols, are preserved in all our examples;
2. Shows that, while some versions of scrubbing functions based on *volatile data pointer* are secure, it is easy to implement this mechanism incorrectly, i.e. using volatile pointer to non-volatile data, or passing a pointer to volatile in a function call;
3. Interestingly it also shows that scrubbing mechanisms based on *volatile function pointers can introduce additional register spilling* that might break secret-erasure with `gcc -O2` and `gcc -O3`.

Finally this framework is open-source [84] and can be easily extended with new compilers and new scrubbing functions.

**Related background**

The following background is recommended before reading this chapter:

— introduction of symbolic execution, given in Section 2.2,

— definition of information-flow policies and self-composition, given in Section 3.1,

— definition of constant-time, given in Section 3.2.4,

— basics of binary analysis, given in Section 2.3—in particular the low-level language used in this section, defined in Section 2.3.4 and the binary-level symbolic execution on which we build, defined in Section 2.3.5.

This chapter builds on the work introduced in Chapter 4 and is easier to follow after reading Chapter 4. Nevertheless, it can (mostly [a]) be read independently from Chapter 4.

In order to keep this chapter self-sufficient, if details introduced in Chapter 4 are necessary for the comprehension of this chapter, we include them using **reminder** boxes.

*a.* An exception is the extension of the proofs in Section 5.4.3

## 5.2 Motivating example

**Secret-erasure.** Secret-erasure requires to erase secret data from the memory after the execution of a program in order to protect them against subsequent memory disclosure vulnerabilities. It is enforced using *scrubbing functions*—functions that overwrite a given part of the memory with dummy values. Take for instance the program in Listing 5.1. The program (1) reads and stores secret data in a `secret` buffer, (2) performs some computations using this secret data, (3) overwrites the content of the `secret` buffer with 0, using the `memset` function. Assuming that the functions `process_secret` does not copy secret data outside the `secret` buffer, we could consider that this programs correctly enforces secret-erasure. *However, the compiled version of this program does not correctly enforce secret-erasure.*

```
void scrub(char *buf, size_t size) {
    memset(buf, 0, size);
}
int critical_function() {
    char secret[LEN];
    for (int i = 0; i < LEN; ++i) {
      secret[i] = read_secret();
    }
    process_secret(secret, LEN); // do something with secret
    scrub(secret, LEN);          // erase secret from memory
    return 0;
}
```

Listing 5.1 – Example of a critical function (incorrectly) enforcing secret-erasure.

**Secret-erasure at binary-level.** The problem with Listing 5.1 is that the `memset` function does not affect the result of the program. As a consequence—in accordance with the C standard—it can be optimized away by the compiler [269, 43, 101, 83]. Indeed at optimization level `O2`, `gcc` completely removes the call to the `memset` function, leaving secret data in the memory and introducing a violation of secret erasure, as illustrated in Listing 5.2. Unfortunately, the C language does not provides a guaranteed way to enforce secret-erasure, hence programmers have to trick the compiler into keeping their scrubbing functions (some of these mechanisms will be detailed in Section 5.6). Moreover, if there are not enough registers, the compiler can also move data from registers to memory. This process, called *register spilling*, is not predictable at source level and can leave secrets in the memory, violating secret-erasure. In this context, it is crucial to be able to verify *at binary-level* that scrubbing functions are not optimized away and that no secrets remain in the memory at the end of a program.

```
1    [...] // saving context
2    jmp     condition
3  loop:
4    call    read_secret
5    store   @secret+i, al // secret[i] = read_secret
6    add     i, 1 // i++
7  condition:
8    ite (i < LEN) goto loop else goto next
9  next:
10   [...] // stack operations
11   push    LEN
12   push    @secret
13   call    process_secret // process_secret(secret, LEN);
14   [...] // restoring context
15   ret
16   jmp
```

LISTING 5.2 – Compiled version (simplified) of the program in Listing 5.1, using `gcc -O2`.

**Our proposal.** In Section 5.3, we propose a definition of secret-erasure for binary-level code in order to enforce secret-erasure without relying on the compiler to preserve the property. In Section 5.4, we propose a symbolic analysis that encompasses our definition of secret-erasure and add support for secret-erasure in BINSEC/REL. Finally, in Section 5.6, we evaluate several scrubbing functions in multiple compiler settings, highlighting several secure and insecure scrubbing mechanisms.

## 5.3 Concrete semantics and leakage model

**Section overview**

In this section, we extend the leakage model introduced in Section 4.3.1 to capture—in addition to constant-time—a larger set of information-flow properties (cf. Section 5.3.1); and define a property for secret-erasure (cf. Section 5.3.2).

### 5.3.1 Leakage model

The leakage model is expressed in the DBA language [28], introduced in Section 2.3.4. As in Section 4.3.1, the behavior of the program is modeled with an operational semantics where transition are labeled with an explicit notion of leakage. Building on Barthe, Grégoire, and Laporte's framework [35], our semantics is parameterized with leakage functions, which permits to consider several leakage models—whereas the leakage model defined in Section 4.3.1 was fixed.

The set of program leakages, denoted $\mathcal{L}$, is defined according to the leakage model. A transition from a configuration $c$ to a configuration $c'$ produces a leakage $t \in \mathcal{L}$, denoted $c \xrightarrow[t]{} c'$. Analogously, the evaluation of an expression $e$ in a configuration $(l, r, m)$, produces a leakage $t \in \mathcal{L}$, denoted $(l, r, m)\ e \vdash_t bv$.

**Reminder (cf. Section 4.3.1)**

The leakage of a multistep execution is the concatenation of leakages, denoted $\cdot$, produced by individual steps. We use $\xrightarrow[t]{}^{k}$ with $k$ a natural number to denote $k$ steps in the concrete semantics.

The concrete semantics is given in Figure 5.1 and is parameterized with leakage functions $\lambda_{\blacktriangleleft} : BV \to \mathcal{L}$, $\lambda_{\blacklozenge} : BV \times BV \to \mathcal{L}$, $\lambda_{@} : BV_{32} \to \mathcal{L}$, $\lambda_{pc} : Loc \to \mathcal{L}$, $\lambda_{\bot} : Loc \to \mathcal{L}$, $\lambda_{\mu} : BV_{32} \times BV_{8} \to \mathcal{L}$. Section 2.3.4 gives a functional explanation of the rules, we focus here on the leakage. A *leakage model* is an instantiation of the leakage functions. We consider the *program counter*, *memory obliviousness*, *size noninterference* and *constant-time*, leakage models defined in [35]. In addition, we define the *operand noninterference* and *secret-erasure* leakage models.

**Program counter [35].** The program counter leakage model leaks the control flow of the program. The leakage of a program is a list of program location: $\mathcal{L} \triangleq List(Loc)$. The outcome of conditionals and the address of indirect jumps is leaked: $\lambda_{pc}(l) = [l]$. We also leak the location of the end of the program: $\lambda_{\bot}(l) = [l]$. Other instructions produce an empty leakage.

**Memory obliviousness [35].** The memory obliviousness leakage model leaks the sequence of memory addresses accessed along the execution. The leakage of a program is a list of 32-bit bitvectors representing addresses of memory accesses: $\mathcal{L} \triangleq List(BV_{32})$. The addresses of memory load and stores are leaked: $\lambda_{@}(e) = [e]$. Other instructions produce an empty leakage.

**Operand noninterference.** The operand noninterference leakage model leaks the value of operands (or part of it) for specific operators that execute in non constant-time. The leakage of a program is a list of bitvector values: $\mathcal{L} \triangleq List(BV)$. Functions $\lambda_{\blacktriangleleft}$ and $\lambda_{\blacklozenge}$ are defined according to architecture specifics. For instance, in some architectures, the execution time of shift or rotation instructions depends on the shift or rotation count [1]. In this case, we can define $\lambda_{<<}(bv_1, bv_2) = [bv_2]$. Other instructions produce an empty leakage.

**Size noninterference [35].** The size noninterference leakage model is a special case of operand noninterference where the size of the operand is leaked. For instance, knowing that the execution time of the division depends on the size of its operands, we can define $\lambda_{\div}(bv_1, bv_2) = [size(bv_1), size(bv_2)]$.

**Constant-time [35].** The constant-time leakage model combines the program counter and the memory obliviousness security policies. The set of leakage is defined as $\mathcal{L} \triangleq List(Loc \cup BV_{32})$. The control flow is leaked $\lambda_{pc}(l) = [l]$, as well as the memory accesses $\lambda_{@}(e) = [e]$. Other instructions produce an empty leakage. Note that some definitions of constant-time also include size noninterference [35] or operand noninterference [36].

**Secret-erasure.** The secret-erasure policy leaks the index and value of every store operation—values that are overwritten are filtered-out from the leakage trace (as we formalize later in Definition 14). With regard to secret dependent control-flow, we

1. See https://bearssl.org/constanttime.html

**Expr**

$$\text{CST}\ \frac{}{(l,r,m)\ \mathtt{bv} \vdash_{\varepsilon} \mathtt{bv}} \qquad \text{VAR}\ \frac{}{(l,r,m)\ \mathtt{v} \vdash_{\varepsilon} r\ \mathtt{v}}$$

$$\text{UNOP}\ \frac{(l,r,m)\ e \vdash_{t} \mathtt{bv}}{(l,r,m)\ \blacktriangleleft\ e \vdash_{t \cdot \lambda_{\blacktriangleleft}(\mathtt{bv})} \blacktriangleleft\ \mathtt{bv}}$$

$$\text{BINOP}\ \frac{(l,r,m)\ e_1 \vdash_{t_1} \mathtt{bv_1} \qquad (l,r,m)\ e_2 \vdash_{t_2} \mathtt{bv_2}}{(l,r,m)\ e_1 \blacklozenge e_2 \vdash_{t_1 \cdot t_2 \cdot \lambda_{\blacklozenge}(\mathtt{bv_1},\mathtt{bv_2})} \mathtt{bv_1} \blacklozenge \mathtt{bv_2}}$$

$$\text{LOAD}\ \frac{(l,r,m)\ e \vdash_{t} \mathtt{bv}}{(l,r,m)\ \mathtt{load}\ e \vdash_{t \cdot \lambda_{@}(\mathtt{bv})} m\ \mathtt{bv}}$$

**Instr**

$$\text{HALT}\ \frac{\mathbb{P}[l] = \mathtt{halt}}{(l,r,m) \xrightarrow[\lambda_{\perp}(l)]{} (l,r,m)} \qquad \text{S\_JUMP}\ \frac{\mathbb{P}[l] = \mathtt{goto}\ l'}{(l,r,m) \to (l',r,m)}$$

$$\text{I\_JUMP}\ \frac{\mathbb{P}[l] = \mathtt{goto}\ e \qquad (l,r,m)\ e \vdash_{t} \mathtt{bv} \qquad l' \triangleq to\_loc(\mathtt{bv})}{(l,r,m) \xrightarrow[t \cdot \lambda_{pc}(l')]{} (l',r,m)}$$

$$\text{ITE-TRUE}\ \frac{\mathbb{P}[l] = \mathtt{ite}\ e\ \mathtt{?}\ l_1\ \mathtt{:}\ l_2 \qquad (l,r,m)\ e \vdash_{t} \mathtt{bv} \qquad \mathtt{bv} \neq 0}{(l,r,m) \xrightarrow[t \cdot \lambda_{pc}(l_1)]{} (l_1,r,m)}$$

$$\text{ITE-FALSE}\ \frac{\mathbb{P}[l] = \mathtt{ite}\ e\ \mathtt{?}\ l_1\ \mathtt{:}\ l_2 \qquad (l,r,m)\ e \vdash_{t} \mathtt{bv} \qquad \mathtt{bv} = 0}{(l,r,m) \xrightarrow[t \cdot \lambda_{pc}(l_2)]{} (l_2,r,m)}$$

$$\text{ASSIGN}\ \frac{\mathbb{P}[l] = \mathtt{v}\ \mathtt{:=}\ e \qquad (l,r,m) e \vdash_{t} \mathtt{bv}}{(l,r,m) \to (l+1, r[\mathtt{v} \mapsto \mathtt{bv}], m)}$$

$$\text{STORE}\ \frac{\mathbb{P}[l] = \mathtt{store}\ e\ e' \qquad (l,r,m)\ e \vdash_{t} \mathtt{bv} \qquad (l,r,m)\ e' \vdash_{t'} \mathtt{bv'}}{(l,r,m) \xrightarrow[t' \cdot t \cdot \lambda_{@}(\mathtt{bv}) \cdot \lambda_{\mu}(\mathtt{bv},\mathtt{bv'})]{} (l+1, r, m[\mathtt{bv} \mapsto \mathtt{bv'}])}$$

FIGURE 5.1 – Concrete evaluation of DBA instructions and expressions.

define a conservative notion of secret-erasure forbidding to branch on secrets—thus including the program counter policy. The leakage of a program is a list of locations and pairs of bitvector values: $\mathcal{L} \triangleq List(Loc \cup (BV_{32} \times BV_8))$. The control flow is leaked $\lambda_{pc}(l) = [l]$, as well as the end of the program $\lambda_{\perp}(l) = [l]$, and the list of store operations $\lambda_{\mu}(bv, bv') = [(bv, bv')]$. Other instructions produce an empty leakage.

### 5.3.2 Secure program

**Reminder (cf. Section 4.3.2)**

Let $\mathcal{V}_h \subseteq \mathcal{V}$ be the set of high (secret) variables and $\mathcal{V}_l = \mathcal{V} \setminus \mathcal{V}_h$ be the set of low (public) variables. Analogously, we define $\mathcal{A}_h \subseteq BV_{32}$ (resp. $\mathcal{A}_l = BV_{32} \setminus \mathcal{A}_h$) as the addresses containing high (resp. low) input in the initial memory. The *low-equivalence relation* over concrete configurations $c$ and $c'$, denoted $c \simeq_l c'$, is defined as the equality of low variables and low parts of the memory.

**Definition 11** (Low equivalence of states ($c \simeq_l c'$))**.** *Two configurations $c \triangleq (l, r, m)$, and $c' \triangleq (l', r', m')$ are low-equivalent if and only if:*

- *for all variable $v \in \mathcal{V}_l$, $r\ v = r'\ v$ and,*
- *for all address $a \in \mathcal{A}_l$, $m\ a = m'\ a$.*

Security is expressed as a form of observational noninterference that is parameterized by the leakage model. Intuitively observational noninterference guarantees that low-equivalent configurations produce the same observations, according to the leakage model:

**Definition 12** (Observational noninterference (ONI))**.** *A program is observationally noninterferent if and only if for all low-equivalent initial configurations $c_0$ and $c'_0$, and for all $k \in \mathbb{N}$,*

$$c_0 \simeq_l c'_0 \ \wedge\ c_0 \xrightarrow[t]{}^{k} c_k \ \wedge\ c'_0 \xrightarrow[t']{}^{k} c'_k \implies \mathit{filter}(t) = \mathit{filter}(t')$$

*The property is parameterized by the leakage model and by a function, $\mathit{filter} : \mathcal{L} \to \mathcal{L}$, that further restricts the leakage.*

Notice that, this definition is *termination insensitive*. However, like in Definition 9, if the leakage determines the control-flow—i.e. $\mathit{filter}(t) = \mathit{filter}(t')$ implies that $c_k$ and $c'_k$ are at the same program point—then $c_k$ is in a final configuration (i.e. on a `halt` instruction) if and only if $c'_k$ is in a final configuration. In this case the definition is *termination sensitive*. Notably, this is the case of the constant-time (Definition 13) and secret-erasure (Definition 14) properties that we consider in this chapter.

**Definition 13** (Constant-time)**.** *A program is constant-time (CT) if it is ONI in the constant-time leakage model with filter set to the identity function.*

Intuitively, our definition of secret erasure guarantees that every secret data that have been stored into the memory must have been erased by the end of the program:

**Definition 14** (Secret-erasure)**.** *A program enforces secret-erasure if it is ONI in the secret-erasure leakage model with filter set to the identity function for control-flow leakages and only leaking store values at the end of the program ($l \in Loc_{\perp}$), restricting to values that have not been overwritten by a more recent store. Formally,*

*filter*$(t) = filter'(t, acc)$ *where* $acc$ *is an initially empty partial function from* $BV_{32}$ *to* $BV_8$ *and* $filter'(t, acc)$ *is defined as:*

$$filter'(\varepsilon, acc) = \varepsilon \tag{5.1}$$

$$filter'((\mathtt{a}, \mathtt{v}) \cdot t, acc) = filter'(t, acc[\mathtt{a} \mapsto \mathtt{v}]) \tag{5.2}$$

$$filter'(l \cdot t, acc) = \begin{cases} acc(\mathtt{a_0}) \cdot \ldots \cdot acc(\mathtt{a_n}), \text{ for } \mathtt{a_i} \in dom(acc) & \text{if } l \in Loc_\perp \\ l \cdot filter(t, acc) & \text{otherwise} \end{cases} \tag{5.3}$$

In Eq. (5.2), *filter'* accumulates store operations $(\mathtt{a}, \mathtt{c})$, from the leakage trace $t$, in the function $acc$. Notice that because $acc$ is a function, if $acc(\mathtt{a})$ is already defined, its value will be replaced by $\mathtt{v}$ after $acc[\mathtt{a} \mapsto \mathtt{v}]$. In Eq. (5.3), if $l$ is a terminal location, all the store values accumulated in $acc$ are leaked. Otherwise, $l$ is a control-flow label and is added to the final leakage trace. An example of application secret-erasure is given in Example 11.

**Example 11** (Secret-erasure)**.** We illustrate our definition of secret-erasure on the insecure low-level program given in Listing 5.2 with `LEN=2`. This program contains three instructions producing a leakage:

- At line 5, the `store` instruction leaks the address and the value of the store;
- At line 8, the conditional jump leaks its next target (`loop` if `i < LEN` and line 9 otherwise);
- At line 15, the current location is leaked, indicating the end of the program.

Consider two low-equivalent executions of this program such that the function `read_secret` returns the sequence $\{\mathtt{a} \cdot \mathtt{b}\}$ in the first execution and $\{\mathtt{c} \cdot \mathtt{d}\}$ in the second execution. At the end of the program, we obtain the following leakage traces (where line numbers denote locations returned by $\lambda_{pc}$):

$$filter(line\ 3 \cdot (\texttt{@secret+0}, \mathtt{a}) \cdot line\ 3 \cdot (\texttt{@secret+1}, \mathtt{b}) \cdot line\ 9 \cdot line\ 15)$$

$$filter(line\ 3 \cdot (\texttt{@secret+0}, \mathtt{c}) \cdot line\ 3 \cdot (\texttt{@secret+1}, \mathtt{d}) \cdot line\ 9 \cdot line\ 15)$$

Because line 15 is the end of the program, the store values are not filtered away, which gives the following traces:

$$[line\ 3 \cdot \mathtt{a} \cdot line\ 3 \cdot \mathtt{b} \cdot line\ 9 \cdot line\ 15)] \neq [line\ 3 \cdot \mathtt{c} \cdot line\ 3 \cdot \mathtt{d} \cdot line\ 9 \cdot line\ 15]$$

Because these leakage traces are distinct, there is a *violation of secret-erasure*.

Now consider that the `memset` function is called just before line 15, and writes 0 at addresses `@secret+0` and `@secret+1`. At the end of the program, we obtain the following leakage traces (omitting control-flow leakage for clarity):

$$filter((\texttt{@secret+0}, \mathtt{a}) \cdot (\texttt{@secret+1}, \mathtt{b}) \cdot (\texttt{@secret+0}, 0) \cdot (\texttt{@secret+1}, 0) \cdot line\ 15)$$

$$filter((\texttt{@secret+0}, \mathtt{c}) \cdot (\texttt{@secret+1}, \mathtt{d}) \cdot (\texttt{@secret+0}, 0) \cdot (\texttt{@secret+1}, 0) \cdot line\ 15)$$

Because *filter* accumulates store operations in a function $acc$, previous writing to `@secret+0` and `@secret+1` are replaced by 0, giving the following traces:

$$[0 \cdot 0 \cdot line\ 15] = [0 \cdot 0 \cdot line\ 15]$$

Because both traces are equal, there is *no violation of secret-erasure*.

Notice that this definition of secret-erasure is conservative because it forbids secret-dependent control-flow. Take for instance the programs in Listing 5.3. Both of these programs will be considered insecure by BINSEC/REL because they contain secret-dependent conditions. In the case of Listing 5.3a, this effectively prevents an implicit flow. However, in the case of Listing 5.3b, we could consider the program secure since the secret is effectively erased from memory but our definition rejects it.

```
if h
then store l 0
else store l 1
```

(A) Secret-dependent condition *with* implicit flow.

```
if h
then store l 0
else store l 0
```

(B) Secret-dependent condition *without* implicit flow.

LISTING 5.3 – Examples of programs where `h` is a high variable and `l` is a low variable.

## 5.4 Parameterized relational symbolic execution

**Section overview**

This section presents the technical contributions of this chapter:

- First, it presents our symbolic evaluation. This is an extension of the binary-level RelSE defined in Section 4.4.1, that is parameterized with *symbolic leakage predicates* to account for the leakage functions introduced in Section 5.3.1 (cf. Section 5.4.1);
- Then it instantiates symbolic leakage predicates according to concrete leakage functions defined in Section 5.3.1 (cf. Section 5.4.2);
- Finally, it discusses the adaption of the theorems introduced in Section 4.4.3 (cf. Section 5.4.3).

### 5.4.1 Parameterized symbolic evaluation

**Reminder (cf. Section 2.3.5)**

Binary-level symbolic execution relies on the quantifier-free theory of fixed-size bitvectors and arrays (QF_ABV [29]). We let $\Phi$ denote the set of symbolic expressions in the QF_ABV logic and $\varphi$, $\phi$, $\psi$, $\iota$ be symbolic expressions ranging over $\Phi$.

A model $M$ assigns concrete values to symbolic variables. The satisfiability of a formula $\pi$ with a model $M$ is denoted $M \vDash \pi$. In the implementation, an SMT solver is used to determine satisfiability of a formula and obtain a satisfying model, denoted $M \vDash_{\text{SMT}} \pi$. Whenever the model is not needed for our purposes, we leave it implicit and simply write $\vDash \pi$ or $\vDash_{\text{SMT}} \pi$ for satisfiability.

**Reminder (cf. Section 4.4.1)**

A *relational* expression $\widehat{\varphi}$ is either a simple symbolic expression $\langle \varphi \rangle$ or a pair $\langle \varphi_l \mid \varphi_r \rangle$ of two symbolic expression in $\Phi$. We denote $\widehat{\varphi}_{|l}$ (resp. $\widehat{\varphi}_{|r}$), the projection

on the left (resp. right) value of $\widehat{\varphi}$. If $\widehat{\varphi} = \langle\varphi\rangle$, then $\widehat{\varphi}_{|l}$ and $\widehat{\varphi}_{|r}$ are both defined as $\varphi$. Let $\mathbf{\Phi}$ be the set of relational formulas and $\boldsymbol{\mathcal{B}v_n}$ be the set of relational symbolic bitvectors of size $n$.

**Reminder (cf. Section 4.4.1.2)**

**Symbolic configuration.** Our symbolic evaluation restricts to pairs of traces following the same path—which is sufficient for constant-time and our definition of secret-erasure because two low-equivalent executions must have the same control-flow. Therefore, a symbolic configuration only needs to consider a single program location $l \in \mathit{Loc}$ at any point of the execution. A *symbolic configuration* is of the form $\left(l, \rho, \widehat{\mu}, \pi\right)$ where:

- $l \in \mathit{Loc}$ is the current program point,
- $\rho : \mathcal{V} \to \mathbf{\Phi}$ is a symbolic register map, mapping variables from a set $\mathcal{V}$ to their symbolic representation as a relational expression in $\mathbf{\Phi}$,
- $\widehat{\mu} : (\mathit{Array}\ \mathcal{B}v_{32}\ \mathcal{B}v_8) \times (\mathit{Array}\ \mathcal{B}v_{32}\ \mathcal{B}v_8)$ is the symbolic memory—a pair of arrays of values in $\mathcal{B}v_8$ indexed by addresses in $\mathcal{B}v_{32}$,
- $\pi \in \Phi$ is the path predicate—a conjunction of conditional statements and assignments encountered along a path.

The location $l$ is not needed for symbolic evaluation of expressions, therefore, we omit it and write $\left(\rho, \mu, \pi\right)$ to denote a symbolic configuration in this context.

**Symbolic evaluation.** The symbolic evaluation of instructions, denoted $s \rightsquigarrow s'$ where $s$ and $s'$ are symbolic configurations, is given in Figure 5.3. The evaluation of an expression *expr* in a state $\left(\rho, \widehat{\mu}, \pi\right)$ to a relational formula $\widehat{\varphi}$, is denoted $\left(\rho, \widehat{\mu}, \pi\right)\ \mathit{expr} \vdash \widehat{\varphi}$ and is given in Figure 5.2. For simplicity, the rules are presented without the optimizations for binary-level RelSE introduced in Section 4.4.2. However these optimizations are still applicable in this context.

**Symbolic leakage predicates.** The symbolic evaluation is parameterized by *symbolic leakage predicates* (in boxes) which are instantiated according to the leakage model (details on the instantiation will be given in Section 5.4.2). Symbolic leakage predicates take as input a path predicate and expressions that can be leaked, and return true if and only if no secret data can leak. The rules of the symbolic evaluation are guarded by these symbolic leakage predicates: a rule can only be evaluated if the associated leakage predicate evaluates to *true*, if it evaluates to $\mathit{false}$ then a leak is detected. Detailed explanations of the symbolic evaluation rules follow:

- $\tilde{\lambda}_{\blacktriangleleft} : \Phi \times \boldsymbol{\mathcal{B}v} \to \mathit{Bool}$ allows for leaking information on the operand of a unary operation $\blacktriangleleft$ in rule UNOP,
- $\tilde{\lambda}_{\blacklozenge} : \Phi \times \boldsymbol{\mathcal{B}v} \times \boldsymbol{\mathcal{B}v} \to \mathit{Bool}$ allows for leaking information on the operand of a binary operation $\blacklozenge$ in rule BINOP,
- $\tilde{\lambda}_{@} : \Phi \times \boldsymbol{\mathcal{B}v} \to \mathit{Bool}$ allows for leaking information on the address of a memory access in rules LOAD and STORE,
- $\tilde{\lambda}_{ij} : \Phi \times \boldsymbol{\mathcal{B}v} \to \mathit{Bool}$ allows for leaking information on the jump target of an indirect jump in rule I-JUMP,
- $\tilde{\lambda}_{ite} : \Phi \times \boldsymbol{\mathcal{B}v} \to \mathit{Bool}$ allows for leaking information on the outcome of a conditional jump in rule ITE-FALSE and ITE-TRUE,

— $\tilde{\lambda}_{\perp} : \Phi \times (Array\ \mathcal{B}v_{32}\ \mathcal{B}v_8) \times (Array\ \mathcal{B}v_{32}\ \mathcal{B}v_8) \to Bool$ allows for leaking information on the memory in rule HALT,

**Expr**

$$\text{CST}\ \frac{}{(\rho,\widehat{\mu},\pi)\ \texttt{bv} \vdash \langle bv \rangle} \qquad \text{VAR}\ \frac{}{(\rho,\widehat{\mu},\pi)\ \texttt{v} \vdash \rho\ \texttt{v}}$$

$$\text{UNOP}\ \frac{(\rho,\widehat{\mu},\pi)\ e \vdash \widehat{\phi} \qquad \widehat{\varphi} \triangleq \blacktriangleleft\, \widehat{\phi} \qquad \boxed{\tilde{\lambda}_{\blacktriangleleft}(\pi,\widehat{\phi})}}{(\rho,\widehat{\mu},\pi) \blacktriangleleft e \vdash \widehat{\varphi}}$$

$$\text{BINOP}\ \frac{(\rho,\widehat{\mu},\pi)\ e_1 \vdash \widehat{\phi} \qquad (\rho,\widehat{\mu},\pi)\ e_2 \vdash \widehat{\psi} \qquad \widehat{\varphi} \triangleq \widehat{\phi} \diamond \widehat{\psi} \qquad \boxed{\tilde{\lambda}_{\blacklozenge}(\pi,\widehat{\phi},\widehat{\psi})}}{(\rho,\widehat{\mu},\pi)\ e_1 \blacklozenge e_2 \vdash \widehat{\varphi}}$$

$$\text{LOAD}\ \frac{(\rho,\widehat{\mu},\pi)\ e_{idx} \vdash \hat{\imath} \qquad \widehat{\varphi} \triangleq \langle select(\widehat{\mu}_{|l},\hat{\imath}_{|l}) \,|\, select(\widehat{\mu}_{|r},\hat{\imath}_{|r}) \rangle \qquad \boxed{\tilde{\lambda}_{@}(\pi,\hat{\imath})}}{(\rho,\widehat{\mu},\pi)\ \texttt{load}\ e_{idx} \vdash \widehat{\varphi}}$$

FIGURE 5.2 – Symbolic evaluation of DBA expressions where $\blacktriangleleft$ (resp. $\diamond$) is the logical counterpart of the concrete operator $\blacktriangleleft$ (resp. $\blacklozenge$).

### 5.4.2 Instantiation of leakage predicates

Symbolic leakage predicates are instantiated according to concrete leakage models defined in Section 5.3.1. Note that the analysis can easily be extended to other leakage models by defining symbolic leakage predicates accordingly.

**Reminder (cf. Section 4.4.1.1)**

For the security evaluation, we define a predicate $secLeak : \mathbf{\Phi} \times \Phi \to Bool$, which ensures that a relational formula does not differ in its right and left components, meaning that it can be leaked securely:

$$secLeak(\widehat{\varphi},\pi) = \begin{cases} true & \text{if } \widehat{\varphi} = \langle \varphi \rangle \\ true & \text{if } \widehat{\varphi} = \langle \varphi_l \,|\, \varphi_r \rangle \wedge \not\models_{\text{SMT}} \pi \wedge \varphi_l \neq \varphi_r \\ false & \text{otherwise} \end{cases}$$

**Program counter.** In the program counter leakage model, symbolic leakage predicates ensure that the outcome of conditional instructions and the addresses of indirect jumps are the same in both executions: $\tilde{\lambda}_{ij}(\widehat{\varphi}) = secLeak(\widehat{\varphi},\pi)$ and $\tilde{\lambda}_{ite}(\widehat{\varphi}) = secLeak(\boldsymbol{eq_0}\ \widehat{\varphi},\pi)$ where $eq_0\ x$ returns $true$ if $x = 0$ and $false$ otherwise, and $\boldsymbol{eq_0}$ is the lifting of $eq_0$ to relational formulas. Other symbolic leakage predicates evaluate to true.

**Memory obliviousness.** In the memory obliviousness leakage model, symbolic leakage predicates ensure that store and load indexes are the same in both executions: $\tilde{\lambda}_{@}(\widehat{\varphi}) = secLeak(\widehat{\varphi},\pi)$. Other symbolic leakage predicates evaluate to true.

**Instr**

$$\text{HALT}\ \frac{\mathbb{P}[l] = \texttt{halt} \qquad \boxed{\tilde{\lambda}_{\perp}(\pi, \widehat{\mu})}}{\left(l, \rho, \widehat{\mu}, \pi\right) \rightsquigarrow \left(l, \rho, \widehat{\mu}, \pi\right)}$$

$$\text{S\_JUMP}\ \frac{\mathbb{P}[l] = \texttt{goto}\ l'}{\left(l, \rho, \widehat{\mu}, \pi\right) \rightsquigarrow \left(l', \rho, \widehat{\mu}, \pi\right)}$$

$$\text{I\_JUMP}\ \frac{\begin{array}{c}\mathbb{P}[l] = \texttt{goto}\ e \\ \left(\rho, \widehat{\mu}, \pi\right)\ e \vdash \widehat{\varphi} \qquad M \vDash_{\text{SMT}} \pi \wedge \widehat{\varphi}_{|l} = \widehat{\varphi}_{|r} \qquad l' \triangleq to\_loc(M(\widehat{\varphi}_{|l})) \\ \pi' \triangleq \pi \wedge (\widehat{\varphi}_{|l} = \widehat{\varphi}_{|r} = M(\widehat{\varphi}_{|l})) \qquad \boxed{\tilde{\lambda}_{ij}(\pi, \widehat{\varphi})}\end{array}}{\left(l, \rho, \widehat{\mu}, \pi\right) \rightsquigarrow \left(l', \rho, \widehat{\mu}, \pi'\right)}$$

ITE-TRUE

$$\frac{\begin{array}{c}\mathbb{P}[l] = \texttt{ite}\ e\ \texttt{?}\ l_{true}\ \texttt{:}\ l_{false} \qquad l' \triangleq l_{true} \\ \left(\rho, \widehat{\mu}, \pi\right)\ e \vdash \widehat{\varphi} \qquad \pi' \triangleq \pi \wedge (\widehat{\varphi}_{|l} \neq 0) \wedge (\widehat{\varphi}_{|r} \neq 0) \qquad \vDash_{\text{SMT}} \pi' \qquad \boxed{\tilde{\lambda}_{ite}(\pi, \widehat{\varphi})}\end{array}}{\left(l, \rho, \widehat{\mu}, \pi\right) \rightsquigarrow \left(l', \rho, \widehat{\mu}, \pi'\right)}$$

ITE-FALSE

$$\frac{\begin{array}{c}\mathbb{P}[l] = \texttt{ite}\ e\ \texttt{?}\ l_{true}\ \texttt{:}\ l_{false} \qquad l' \triangleq l_{false} \\ \left(\rho, \widehat{\mu}, \pi\right)\ e \vdash \widehat{\varphi} \qquad \pi' \triangleq \pi \wedge (\widehat{\varphi}_{|l} = 0) \wedge (\widehat{\varphi}_{|r} = 0) \qquad \vDash_{\text{SMT}} \pi' \qquad \boxed{\tilde{\lambda}_{ite}(\pi, \widehat{\varphi})}\end{array}}{\left(l, \rho, \widehat{\mu}, \pi\right) \rightsquigarrow \left(l', \rho, \widehat{\mu}, \pi'\right)}$$

$$\text{ASSIGN}\ \frac{\begin{array}{c}\mathbb{P}[l] = \texttt{v := } e \qquad \left(\rho, \widehat{\mu}, \pi\right)\ e \vdash \widehat{\varphi} \\ \widehat{\varphi}' \triangleq \mathit{fresh}(\widehat{\varphi}) \qquad \rho' \triangleq \rho[v \mapsto \widehat{\varphi}'] \qquad \pi' \triangleq \pi \wedge (\widehat{\varphi}'_{|l} = \widehat{\varphi}_{|l}) \wedge (\widehat{\varphi}'_{|r} = \widehat{\varphi}_{|r})\end{array}}{\left(l, \rho, \widehat{\mu}, \pi\right) \rightsquigarrow \left(l + 1, \rho', \widehat{\mu}, \pi'\right)}$$

STORE

$$\frac{\begin{array}{c}\mathbb{P}[l] = \texttt{store}\ e_{idx}\ e_{val} \qquad \left(\rho, \widehat{\mu}, \pi\right)\ e_{idx} \vdash \hat{\imath} \\ \left(\rho, \widehat{\mu}, \pi\right)\ e_{val} \vdash \hat{\nu} \qquad \widehat{\mu}' \triangleq \langle \mathit{store}(\widehat{\mu}_{|l}, \hat{\imath}_{|l}, \hat{\nu}_{|l}) \,|\, \mathit{store}(\widehat{\mu}_{|r}, \hat{\imath}_{|r}, \hat{\nu}_{|r}) \rangle \\ \pi' \triangleq \pi \wedge \widehat{\mu}'_{|l} = \mathit{store}(\widehat{\mu}_{|l}, \hat{\imath}_{|l}, \hat{\nu}_{|l}) \wedge \widehat{\mu}'_{|r} = \mathit{store}(\widehat{\mu}_{|r}, \hat{\imath}_{|r}, \hat{\nu}_{|r}) \qquad \boxed{\tilde{\lambda}_{@}(\pi, \hat{\imath})}\end{array}}{\left(l, \rho, \widehat{\mu}, \pi\right) \rightsquigarrow \left(l + 1, \rho, \widehat{\mu}', \pi'\right)}$$

FIGURE 5.3 – Symbolic evaluation of DBA instructions where where $\mathit{fresh}(\widehat{\varphi})$ returns a pair of fresh symbolic variables if $\widehat{\varphi}$ is a pair, or a simple fresh symbolic variable if $\widehat{\varphi}$ is simple.

**Operand noninterference.** In the operand noninterference leakage model, symbolic leakage predicates ensure that operands (or part of them) are the same in both executions for specific operators that execute in non constant-time. For instance, for architectures in which the execution time of shift depends on the shift count, $\tilde{\lambda}_{<<}(\widehat{\varphi}, \widehat{\phi}) = \mathit{secLeak}(\widehat{\phi}, \pi)$. Other symbolic leakage predicates evaluate to true.

**Size noninterference** The size noninterference leakage model is a special case of operand noninterference. Symbolic leakage predicates ensure that the size of operands is the same in both executions for specific operators that execute in non constant-time. For instance for the division, we have $\tilde{\lambda}_{\div}(\widehat{\varphi}, \widehat{\psi}) = \mathit{secLeak}(\boldsymbol{size}\widehat{\varphi}, \pi)$, where

$size : \mathcal{B}v \to \mathcal{B}v$ is a function that returns the size of a symbolic bitvector and $\boldsymbol{size}$ its lifting to relational expressions. Other symbolic leakage predicates evaluate to true.

**Constant-time.** The constant-time leakage model is a combination of the program counter and the memory obliviousness leakage models. Symbolic leakage predicates $\tilde{\lambda}_{ij}$ and $\tilde{\lambda}_{ite}$ are defined like in the program counter policy, while $\tilde{\lambda}_{@}$ is defined like in the memory obliviousness policy. Other symbolic leakage predicates evaluate to true.

**Secret-erasure.** In the secret-erasure leakage model, a symbolic leakage predicate ensures, at the end of the program, that the parts of memory that have been written by the program are the same in both executions:

$$\tilde{\lambda}_{\perp}(\widehat{\mu}) = \bigwedge_{\iota \in addr(\widehat{\mu})} secLeak(\langle select(\widehat{\mu}_{|l}, \iota) \,|\, select(\widehat{\mu}_{|r}, \iota)\rangle, \pi)$$

where $addr(\widehat{\mu})$ is the list of store indexes in the symbolic memory $\widehat{\mu}$. An example is given in Example 12. Additionally, symbolic leakage predicates $\tilde{\lambda}_{ij}$ and $\tilde{\lambda}_{ite}$ are defined like in the program counter policy.

**Example 12** (Secret-erasure)**.** We illustrate our analysis for secret-erasure on the insecure low-level program given in Listing 5.2 with `LEN=2`. For simplicity, we ignore control-flow leakages and focus on the leakage from the HALT rule. The important rules to consider in this example are:

— The STORE rule, which updates the symbolic memory with a symbolic *store* operation—and is applied when the `store` instruction at line 5 is evaluated;

— The HALT rule, which leaks the content of the symbolic memory—and is applied when the `ret` instruction at line 15 is evaluated.

Consider the relational symbolic execution of this program such that `read_secret` returns pairs of symbolic expressions $\langle \alpha_l \,|\, \alpha_r \rangle$ and $\langle \beta_r \,|\, \beta_r \rangle$ At the end of the program, we obtain the following symbolic memory (which we represent as history of relational store operations for simplicity):

$$\widehat{\mu} = \boldsymbol{store}(\texttt{@secret+0}, \langle \alpha_l \,|\, \alpha_r \rangle) \cdot \boldsymbol{store}(\texttt{@secret+1}, \langle \beta_l \,|\, \beta_r \rangle)$$

Finally, the rule HALT evaluates the symbolic leakage predicate $\tilde{\lambda}_{\perp}(\widehat{\mu})$ (we omit the path predicate as it does not change the satisfiability of the solver calls):

$$\begin{aligned}
\tilde{\lambda}_{\perp}(\widehat{\mu}) &= secLeak(\boldsymbol{select}(\widehat{\mu}, \texttt{@secret+0})) \wedge secLeak(\boldsymbol{select}(\widehat{\mu}, \texttt{@secret+1})) \\
&= secLeak(\langle \alpha_l \,|\, \alpha_r \rangle) \wedge secLeak(\langle \beta_r \,|\, \beta_l \rangle) \\
&= \not\models_{\text{SMT}} \alpha_l \neq \alpha_r \vee \beta_l \neq \beta_r \\
&= false
\end{aligned}$$

Because $\tilde{\lambda}_{\perp}(\widehat{\mu})$ is unsatisfiable, our analysis reports a *violation of secret-erasure*, together with a counterexample (e.g. $\alpha_l = \texttt{a}$ and $\alpha_r = \texttt{c}$).

Consider now that the `memset` function is called just before line 15, and writes 0 at addresses `@secret+0` and `@secret+1`. At the end of the program, we obtain the following symbolic memory:

$$\begin{aligned}
\widehat{\mu} = \boldsymbol{store}(\texttt{@secret+0}, \langle \alpha_l \,|\, \alpha_r \rangle) \cdot \boldsymbol{store}(\texttt{@secret+1}, \langle \beta_l \,|\, \beta_r \rangle) \cdot \\
\boldsymbol{store}(\texttt{@secret+0}, \langle 0 \rangle) \cdot \boldsymbol{store}(\texttt{@secret+1}, \langle 0 \rangle)
\end{aligned}$$

and the symbolic leakage predicate in the rule HALT becomes:

$$\begin{aligned}\tilde{\lambda}_{\perp}(\widehat{\mu}) &= secLeak(\textbf{\textit{select}}(\widehat{\mu}, \texttt{@secret+0})) \wedge secLeak(\textbf{\textit{select}}(\widehat{\mu}, \texttt{@secret+1})) \\ &= secLeak(\langle 0 \rangle) \wedge secLeak(\langle 0 \rangle) \\ &= true\end{aligned}$$

Because the symbolic leakage predicate is *true*, there is *no violation of secret-erasure*.

### 5.4.3 Adapting theorems and proofs for other leakage models

**Section overview**

This section discusses how the theorems and proofs given in Section 4.4.3 for the constant-time property can be adapted to other leakage models. To avoid unnecessary repetitions, we do not include here the definitions of the theorems and refer the reader to Section 4.4.3.

**Correctness.** Correctness of our symbolic execution (Theorem 1) holds regardless of the leakage model considered. Indeed, we showed that our symbolic execution makes no over-approximation, without using the leakage model for constant-time. Moreover, we can show that (Theorem 1) still holds for other leakage models because symbolic leakage predicates cannot remove constraints from the symbolic state (and therefore cannot introduce over-approximations).

**Bug-Finding.** Bug-finding (Theorem 2) can also be easily adapted to other leakage models as long as the symbolic leakage model does not over-approximate the concrete leakage model. In particular, it still holds for secret-erasure. The adaptation of Theorem 2 to secret-erasure only requires to show that Lemma 1 holds for the HALT rule.

**Completeness.** Completeness (Theorem 3) follows from Lemma 2 and Theorem 2 and thus can be adapted to other leakage models on two conditions. First, because our symbolic semantics is blocking on errors, it only applies to secure programs and its proof relies on the absence of false alarm—which is given as long as the symbolic leakage model does not over-approximate the concrete leakage model (Theorem 2). Second, Lemma 2 only applies to pairs of concrete executions following the same path. Therefore, Theorem 3 only holds for leakage models leaking the control-flow (i.e. that include the program counter leakage model). Note that these two conditions are met in the case of secret-erasure.

**Bounded-verification.** Bounded-verification (Theorem 4) can be adapted to other leakage models on two conditions. First, because it builds on Lemma 2 which only applies to pairs of concrete executions following the same path, it only holds for leakage models leaking the control-flow (i.e. that include the program counter leakage model). Second, it requires to show that the symbolic leakage model does not under-approximate the concrete leakage model: if a leakage occurs in concrete execution then this leakage is captured in symbolic execution. These conditions hold for our definition of secret-erasure, we must just adapt the proof for the HALT rule as the *filter* function delays the leakage of store values upon termination.

## 5.5 Implementation

**Reminder (cf. Section 4.5)**

Relational symbolic evaluation is implemented on top of BINSEC in the `Rel` plugin. It is composed of a *relational symbolic exploration* module and an `Insecurity` module. The symbolic exploration module chooses a path to explore, updates the symbolic state and the path predicate according to the current instruction, and ensures that paths are satisfiable. The `Insecurity` module builds insecurity queries and ensures that they are not satisfiable.

We adapt BINSEC/REL to be modular in the property to check. We transform the `Insecurity` module into a functor that can be parameterized by a `Property` module. The `Property` module defines the property to check and its implementation depends on the leakage model. In particular, it is in charge of:

— collecting expressions that can leak according to the leakage model,

— build insecurity queries, send them to the solver and return counterexamples.

Finally, we implement two `Property` modules:

— A module for constant-time (taken from our prior implementation), which collects addresses of load and store instructions as well as conditions and indirect jump expressions and send them to the solver according to the fault-packing parameter (see Section 4.4.2.3);

— A module for secret-erasure, which collects addresses of store instructions encountered along symbolic execution, and leaks the content of the memory at these addresses at the end of the program.

## 5.6 Application: compiler preservation of secret-erasure

**An extensible framework.** In this section we present a framework to automatically check the preservation of secret-erasure for multiple scrubbing functions and compilers. This framework is open source and *can be easily extended*[2] with *new compilers* and *new scrubbing functions*. Using BINSEC/REL:

— we analyze 17 scrubbing functions;

— with multiple compilers versions of `clang` (3.0, 3.9, 7.1.0, 9.0.1 and 11.0.1) and `gcc` (5.4.0, 6.2.0, 7.2.0, 8.3.0 and 10.2.0);

— and multiple optimization levels (`-O0`, `-O1`, `-O2` and `-O3`).

This accounts for a total of *680 binaries analyzed in 80.1 seconds*, which show that BINSEC/REL can *efficiently* verify secret-erasure on a large number of binaries.

**Setup.** Experiments were performed on a laptop with an Intel(R) Core(TM) i5-2520M CPU @ 2.50GHz processor and 32GB of RAM.

**Legend.** In this section, clang-all-versions (resp. gcc-all-versions) refer to all the aforementioned `clang` (resp. `gcc`) versions; and in tables ✓ indicates that a program is secure and ✗ that it is insecure w.r.t secret-erasure.

2. See https://github.com/binsec/rel_bench/blob/main/src/secret-erasure/Readme.org

### 5.6.1 Naive implementations

First, we consider naive (insecure) implementations of scrubbing functions:

— `loop`: naive scrubbing function that uses a simple `for` loop to set the memory to 0,

— `memset`: uses the `memset` function from the Standard C Library,

— `bzero`: function defined in `glibc` to set memory to 0.

**Results** (cf. Table 5.1). As expected, without appropriate countermeasures, these naive implementation of scrubbing functions are all optimized away by all versions of `clang` and `gcc` at optimization level `-O2` and `-O3`. Additionally, as highlighted in Table 5.1, `bzero` is also optimized away at optimization level `-O1` with `gcc-7.2.0` and older versions [3].

| | clang-3.0 | | | | clang-3.9 | | | | clang-7.1.0 | | | | clang-9.0.1 | | | | clang-11.0.1 | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | O0 | O1 | O2 | O3 | O0 | O1 | O2 | O3 | O0 | O1 | O2 | O3 | O0 | O1 | O2 | O3 | O0 | O1 | O2 | O3 |
| `loop` | ✓ | ✓ | ✗ | ✗ | ✓ | ✓ | ✗ | ✗ | ✓ | ✓ | ✗ | ✗ | ✓ | ✓ | ✗ | ✗ | ✓ | ✓ | ✗ | ✗ |
| `memset` | ✓ | ✓ | ✗ | ✗ | ✓ | ✓ | ✗ | ✗ | ✓ | ✓ | ✗ | ✗ | ✓ | ✓ | ✗ | ✗ | ✓ | ✓ | ✗ | ✗ |
| `bzero` | ✓ | ✓ | ✗ | ✗ | ✓ | ✓ | ✗ | ✗ | ✓ | ✓ | ✗ | ✗ | ✓ | ✓ | ✗ | ✗ | ✓ | ✓ | ✗ | ✗ |
| | gcc-5.4.0 | | | | gcc-6.2.0 | | | | gcc-7.2.0 | | | | gcc-8.3.0 | | | | gcc-10.2.0 | | | |
| | O0 | O1 | O2 | O3 | O0 | O1 | O2 | O3 | O0 | O1 | O2 | O3 | O0 | O1 | O2 | O3 | O0 | O1 | O2 | O3 |
| `loop` | ✓ | ✓ | ✗ | ✗ | ✓ | ✓ | ✗ | ✗ | ✓ | ✓ | ✗ | ✗ | ✓ | ✓ | ✗ | ✗ | ✓ | ✓ | ✗ | ✗ |
| `memset` | ✓ | ✓ | ✗ | ✗ | ✓ | ✓ | ✗ | ✗ | ✓ | ✓ | ✗ | ✗ | ✓ | ✓ | ✗ | ✗ | ✓ | ✓ | ✗ | ✗ |
| `bzero` | ✓ | (✗) | ✗ | ✗ | ✓ | (✗) | ✗ | ✗ | ✓ | (✗) | ✗ | ✗ | ✓ | (✓) | ✗ | ✗ | ✓ | (✓) | ✗ | ✗ |

TABLE 5.1 – Preservation of secret-erasure for naive scrubbing functions.

### 5.6.2 Volatile function pointer

The `volatile` type qualifier indicates that the value of an object may change at any time, preventing the compiler from optimizing memory accesses to volatile objects. This mechanism can be exploited for secure secret-erasure by using a volatile function pointer for the scrubbing function (e.g. eventually redirecting to `memset`). Because the function may change, the compiler cannot optimize it away. Listing 5.4 illustrates the implementation of this mechanism in OpenSSL [193].

```
1 typedef void *(*memset_t)(void *, int, size_t);
2 static volatile memset_t memset_func = memset;
3 void scrub(char *buf, size_t size) {
4   memset_func(buf, 0, size);
5 }
```

LISTING 5.4 – OpenSSL scrubbing function [193]. Relies on volatile function pointer.

3. This is because the function calls to `scrub` and `bzero` are inlined in `gcc-7.2.0` and older versions, making the optimization possible whereas the call to `scrub` is not inlined in `gcc-8.3.0` and older versions.

**Results** (cf. Table 5.2). BINSEC/REL reports that, for all versions of `gcc`, the secret-erasure property is not preserved at optimization levels `-O2` and `-O3`. Indeed, the caller-saved register `edx` is pushed on the stack before the call to the volatile function. However, it contains secret data which are spilled on the stack and not cleared afterwards—note that `clang` also uses `edx` to hold secret data but does not save it on the stack before the function call. *This shows that our tool can find violations of secret erasure from register spilling.* We conclude that while the volatile function pointer mechanism is effective for preventing the scrubbing function to be optimized away, it may also *introduce unnecessary register spilling that might break secret-erasure.*

| clang-all-versions | | | | gcc-all-versions | | | |
|---|---|---|---|---|---|---|---|
| -O0 | -O1 | -O2 | -O3 | -O0 | -O1 | -O2 | -O3 |
| ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ | ✗ |

TABLE 5.2 – Preservation of secret-erasure with volatile function pointer as implemented in Listing 5.4.

### 5.6.3 Volatile data pointer

The volatile type qualifier can also be used for secure secret-erasure by marking the data to scrub as volatile before erasing it. Usages of the volatile qualifier are illustrated in Listing 5.5.

```
1 // Pointer-to-volatile
2 volatile char *vbuf = (volatile char *) buf;
3 // Volatile pointer (pointer is volatile itself)
4 char * volatile vbuf = (char *volatile) buf;
5 // Volatile pointer-to-volatile
6 volatile char *volatile vbuf = (volatile char *volatile) buf;
```

LISTING 5.5 – Different usage of `volatile` type qualifier before scrubbing memory.

We analyze several implementations based on this mechanism:

— `ptr_to_volatile_loop` casts the pointer `buf` to a pointer-to-volatile `vbuf` (cf. Listing 5.5, line 2) before scrubbing data from `vbuf` using a simple `for` or `while` loop. This is a commonly used technique for scrubbing memory, used for instance in Libgcrypt [169], wolfSSL [259], or sudo [181];

— `ptr_to_volatile_memset` is similar to `ptr_to_volatile_loop` but scrubs data from memory using `memset`. Note that this implementation is insecure as the `volatile` type qualifier is discarded by the function call—`volatile char *` is not compatible with `void *`;

— `volatile_ptr_loop` (resp. `volatile_ptr_memset`) casts the pointer `buf` to a volatile pointer `vbuf`—but pointing to non volatile data (cf. Listing 5.5, line 4) before scrubbing data from `vbuf` using a simple `for` or `while` loop (resp. `memset`) [4];

4. Although we did not find this implementation in real-world cryptographic code, we were curious about how the compiler would handle this case.

— `vol_ptr_to_vol_loop` casts the pointer `buf` to a volatile pointer-to-volatile `vbuf` (cf. Listing 5.5, line 6) before scrubbing data from `vbuf` using a simple `for` or `while` loop. It is the fallback scrubbing mechanism used in `libsodium` [170] and in HACL* [133] cryptographic libraries;

— `vol_ptr_to_vol_memset` is similar to `vol_ptr_to_vol_loop` but uses `memset` instead of a `loop`.

**Results** (cf. Table 5.3). First, our experiments show that using *volatile pointers to non-volatile data does not reliably prevent the compiler from optimizing away the scrubbing function.* Indeed, `gcc` optimizes away the scrubbing function at optimization level `-O2` and `-O3` in both `volatile_ptr` implementations. Second, using a pointer to volatile works in the `loop` version (i.e. `ptr_to_volatile_loop` and `vol_ptr_to_vol_loop`) but not in the `memset` versions (i.e. `ptr_to_volatile_memset` and `vol_ptr_to_vol_memset`) as the function call to `memset` discards the volatile qualifier.

| | clang-all-versions | | | | gcc-all-versions | | | |
|---|---|---|---|---|---|---|---|---|
| | -O0 | -O1 | -O2 | -O3 | -O0 | -O1 | -O2 | -O3 |
| ptr_to_volatile_loop | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| volatile_ptr_loop | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ | ✗ |
| vol_ptr_to_vol_loop | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| ptr_to_volatile_memset | ✓ | ✓ | ✗ | ✗ | ✓ | ✓ | ✗ | ✗ |
| volatile_ptr_memset | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ | ✗ |
| vol_ptr_to_vol_memset | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ | ✗ |

TABLE 5.3 – Preservation of secret-erasure with volatile data pointers. ✓ indicates that a program is secure and ✗ that it is insecure.

### 5.6.4 Memory barriers

Memory barriers are inline assembly statements which indicate the compiler that the memory could be read or written, forcing the compiler to preserve preceding store operations. We study four different implementations of memory barriers: three implementations from safeclib [246], plus the approach recommended in a study from Yang et al. on scrubbing functions [269].

— `memory_barrier_simple` (cf. Listing 5.6, line 2) is the implementation used in `explicit_bzero` and the fallback implementation used in safeclib. As pointed by Yang et al. [269], this barrier works with `gcc` [1] but might not work with `clang`, which might optimize away a call to `memset` or a loop before this barrier [54]— although we could not reproduce the behavior in our experiments;

— `memory_barrier_mfence` (cf. Listing 5.6, line 4) is similar to `memory_barrier_simple` with an additional `mfence` instruction for serializing memory. It is used in safeclib when `mfence` instruction is available;

— `memory_barrier_lock` (cf. Listing 5.6, line 6) is similar to `memory_barrier_mfence` but uses a `lock` prefix for serializing memory. It is used in safeclib on `i386` architectures;

— `memory_barrier_ptr` (cf. Listing 5.6, line 8) is a more resilient approach than `memory_barrier_simple`, recommended in the study of Yang et al. [269], and used for instance in libsodium `memzero` [170]. It makes the pointer `buf` visible to

the assembly code, preventing prior store operation to this pointer from being optimized away.

```
1 // memory_barrier_simple
2 __asm__ __volatile__("":::"memory");
3 // memory_barrier_mfence
4 __asm__ __volatile__("mfence" ::: "memory");
5 // memory_barrier_lock
6 __asm__ __volatile__("lock; addl $0,0(%%esp)" ::: "memory");
7 // memory_barrier_ptr
8 __asm__ __volatile__("": :"r"(buf) :"memory");
```

Listing 5.6 – Different implementation of memory barriers.

**Results.** For all the implementation of memory barriers that we tested, we did not find any vulnerability—even with the version deemed insecure in the study of Yang et al. [269]. [5]

### 5.6.5 Weak symbols

Weak symbols are specially annotated symbols (with `__attribute__((weak))`) whose definition may change at link time. An illustration of a weak function symbol is given in Listing 5.7. The compiler cannot optimize a store operation preceding the call to `_sodium_dummy_symbol` because its definition may change and could access the content of the buffer. This mechanism, is used in libsodium `memzero` [170] when weak symbols are available.

```
1 __attribute__((weak)) void _sodium_dummy_symbol(
2   void *const  pnt, const size_t len) {
3     (void) pnt;
4     (void) len;
5 }
6 void scrub(char *buf, size_t size) {
7     memset(buf, 0, size);
8     _sodium_dummy_symbol(buf, size);
9 }
```

Listing 5.7 – Libsodium implementation of memory scrubbing with weak symbols.

**Results.** Binsec/Rel did not find any vulnerability using weak-symbols.

### 5.6.6 Off-the-shelf implementations

Finally, we consider two secure implementations of scrubbing functions proposed in external libraries:

5. As explained in a bug report [54], `memory_barrier_simple` is not reliable because `clang` might consider that the inlined assembly code does not access the buffer (e.g. by fitting all of the buffer in registers). The fact that we were not able to reproduce this bug in our setup is due to differences in programs (in our program the address of the buffer escapes because of function calls whereas it is not the case in the bug report); it does not mean that this barrier is secure (it is not).

— `explicit_bzero` is a function defined in `glibc` to set memory to 0 with a guarantee that it will not be optimized away by the compiler;

— `memset_s` is a function defined in the optional Annex K "bound-checking interfaces" of the C11 standard. We take the implementation of `safeclib` [247], compiled with its default Makefile for a `i386` architecture.

These functions both rely on a memory barrier (see Section 5.6.4) to prevent the compiler from optimizing scrubbing operations.

**Results.** BINSEC/REL did not find any vulnerability with these functions.

## 5.7 Related work

**Section overview**

Related work on symbolic execution for information-flow analysis has already been extensively discussed in Section 4.8, so we focus here on the related work regarding secret-erasure.

**Specification.** Chong and Myers [74] introduce the first framework to specify rich secret-erasure policies which supports secret-erasure and declassification under some conditions. Their policy language is general as it does not impose a particular logic for conditions, nor does it restrict to a specific programming language, however they leave enforcement mechanisms for future work. Tedesco, Hunt, and Sands [239] propose a knowledge-based framework for specifying richer secret-erasure policies, taking into account the observational power of the attacker, the amount of information to erase, and more complex conditions. Askarov et al. [17] additionally considers cryptographic data deletion (i.e. reliably erase data on untrustworthy storage service by encrypting them and deleting the encryption key). These works [74, 239, 17] focus on expressing complex secret-erasure policies, but are not directly applicable to concrete languages.

Hansen and Probst [134] propose the first application of a simple secret-erasure policy for a concrete language (i.e. Java Card Bytecode), which ensures that secrets are unavailable after program termination. Our definition of secret erasure is close to theirs and directly applicable for binary-level verification.

**Verification.** Most enforcement mechanisms for secret-erasure are language-based and rely on type systems to enforce information flow control. This includes type-systems to enforce noninterference and secret-erasure [140], type-based information-flow control in Jif [73] and Java [17], dynamic taint analysis for Python programs [240], and interactive verification in Coq using dependent types [184].

Secretgrind [227] is a dynamic taint tracking tool based on Valgrind [186] that tracks secret data in memory. It is the closest work to ours but is still quite different as their analysis is dynamic and *liberal*, in the sense that it permits implicit flows; whereas our analysis is static and our definition of secret-erasure is *conservative*, in the sense that if forbids secret-dependent control flow (and hence implicit flows). For instance, consider again the programs introduced in Listing 5.3. The program in Listing 5.3a, containing an implicit flow, will be considered insecure by BINSEC/REL but (liberally) considered secure by Secretgrind; whereas the program in Listing 5.3b which contains no implicit nor explicit flow will be considered secure by Secretgrind but (conservatively) considered insecure by BINSEC/REL because it contains a secret-dependent condition.

**Compilation.** The problem of (non-)preservation of secret-erasure by compilers is well known [83, 101, 43, 269, 231]. To remedy it, a notion of information flow-preserving program transformation has been proposed [43] but this approach requires to compile programs using CompCert [168] and does not apply to already compiled binaries. Finally, preservation of secret-erasure functions by compilers has been studied manually [269], and we further this line of work by proposing an extensible framework for *automating* the process.

## 5.8 Conclusion

We generalize binary-level relational symbolic execution to a subset of *information flow* properties, restricting to pairs of traces following the same path, and propose a new leakage model and definition to capture secret-erasure. We adapt, BINSEC/REL, to verify the secret-erasure property and use it to study the preservation of secret-erasure by compilers. We highlight incorrect usages of volatile data pointer for secret erasure, and show that scrubbing mechanisms based on *volatile function pointers* can introduce additional violation from register spilling.

# Chapter 6

# BINSEC/HAUNTED: Symbolic Analyzer for Spectre

**Chapter overview**

Spectre are microarchitectural attacks which were made public in January 2018. They allow an attacker to recover secrets by exploiting speculations in processors. Detection of Spectre is particularly important for cryptographic libraries, and defenses at the software level have been proposed. Yet, defenses correctness and Spectre detection pose challenges due on one hand to the explosion of the exploration space induced by speculative paths, and on the other hand, to the introduction of new Spectre vulnerabilities at different compilation stages.

We propose an optimization, called *Haunted RelSE*, allowing for scalable detection of Spectre vulnerabilities at binary level. We prove the optimization semantically correct with regard to the more naive explicit speculative exploration approach used in state-of-the-art tools. We implement Haunted RelSE in a symbolic analysis tool BINSEC/HAUNTED, and extensively test it on a well-known set of litmus tests for Spectre-PHT, and on a new set of litmus tests for Spectre-STL, which we propose. Our technique finds more violations and scales better than state-of-the-art techniques and tools, analyzing real-world cryptographic libraries and finding new violations. Thanks to our tool, we discover that index-masking—a standard defense for Spectre-PHT—and well-known `gcc` options to compile position independent executable introduce Spectre-STL violations. We propose and verify a correction to index-masking to avoid the problem.

## 6.1 Introduction

Modern CPUs performance relies on complex hardware logic, including out-of-order execution and *speculations*. Independently from the hardware implementation, the *architecture* describes how instructions behave in a CPU and includes state that can be observed by the developer such as data in registers and main memory. The *microarchitecture* describes how the architecture is implemented in a processor hardware, and its state includes for example entries in the cache which are transparent to the developer. In order to improve performance, the CPU can execute instructions ahead of time, and attempt, for instance, to guess values via a branch predictor to speculatively execute a direction of the control flow when the condition is not yet available, instead of stalling the pipeline. If the guess is correct, the execution is committed and the CPU spares the cost of the pipeline stall. If the guess is incorrect,

the CPU discards the speculative execution by reverting the affected state of the architecture. At the end, only correct executions define the state of the architecture. Reverted executions, also known as *transient executions*, are meant to be transparent from the architectural point of view.

Unfortunately, transient executions leave observable microarchitectural side effects that can be exploited by an attacker to recover secrets at the architectural level. This behavior is exploited in *Spectre attacks* [154] which were made public in early 2018. Since then, Spectre attacks have drawn considerable attention from both industry and academy, with works that discovered new Spectre variants [65], new detection methods [129, 253, 67], and new countermesures [26, 148, 176]. To date, there are four known main variants of Spectre attacks [65]: PHT, BTB, RSB, STL, respectively exploiting predictors for conditional branches, indirect branches, return addresses, and store-to-load dependencies.

Most works on analyzers [261, 132, 252, 253, 71, 129] only focus on the Pattern History Table (PHT) variant (a.k.a Spectre-v1 [154]) which exploits conditional branches, yet they struggle on medium-size binary code (cf. Table 6.6). Only one tool, Pitchfork [67], addresses the Store to Load (STL) variant (a.k.a Spectre-v4 [139]), which exploits the memory dependence predictor. Unfortunately, Pitchfork does not scale for analyzing Spectre-STL, even on small programs (cf. Table 6.5). Other variants, RSB and BTB, are currently out-of-scope of static analyzers because modeling them requires to allow an attacker to jump to arbitrary parts of the code [1].

**Goal and challenges.** In this chapter, we propose a novel technique to detect Spectre-PHT and Spectre-STL vulnerabilities and we implement it in a new symbolic analyzer for binary code. Two challenges arise in the design of such an analyzer:

**C1** First, the details of the microarchitecture cannot be fully included in the analysis because they are not public in general and not easy to obtain. Yet the challenge is to find an abstraction powerful enough to capture side channels attacks due to microarchitectural state;

**C2** Second, exploration of all possible speculative executions does not scale because it quickly leads to state explosion. The challenge is how to optimize this exploration in order to make the analysis applicable to real code.

**Proposal.** We tackle challenge **C1** by targeting a relational security property coined in the literature as *speculative constant-time* [67], a property reminiscent of constant-time [35], widely used in cryptographic implementations. Speculative constant-time takes speculative executions into account without explicitly modeling intricate microarchitectural details. However, it is well known that constant-time programming is not necessarily preserved by compilers [231, 87], so our analysis operates at binary level—besides, it is compiler-agnostic and does not require source code. For this, we extend the model of previous work for binary analysis of constant-time presented in Chapter 4 in order to analyze *speculative constant-time* [67].

A well-known analysis technique that scales well on binary code is symbolic execution (SE) [151, 121]. However, in order to analyze speculative constant-time, it must be adapted to consider the speculative behavior of the program. Symbolic analyzers for Spectre-PHT [253, 129, 71] and Spectre-STL [67] model the speculative behavior *explicitly*, by forking the execution to explore transient paths, which quickly leads to state explosion—especially for Spectre-STL which has to fork for each possible

1. Section 7.2 discusses some perspective for future work around these variants, mainly targeting the verification of their countermeasures.

store and load interleaving. The adaptation of symbolic execution to information-flow properties such as constant-time, known as *relational symbolic execution* (RelSE), has proven very successful in terms of scalability and precision for binary level (cf. Chapter 4). In order to address challenge **C2**, our key technical insight is to adapt RelSE to execute transient executions *at the same time* as sequential executions (i.e. executions related to correct speculations). We name this technique *Haunted RelSE*:

— For Spectre-PHT, it prunes redundant states by executing at the same time transient and sequential paths resulting from a conditional statement;

— For Spectre-STL, instead of forking the symbolic execution for each possible load and store interleaving, it prunes redundant cases and encodes the remaining ones in a single symbolic path.

We implement Haunted RelSE in a binary-level symbolic analysis tool called Binsec/Haunted, built on top of Binsec/Rel. For evaluation, we use the well-known Kocher test cases for Spectre-PHT [154], as well as a *new set of test cases that we propose for Spectre-STL*, and real-world cryptographic code from donna, Libsodium and OpenSSL libraries.

**Findings.** Interestingly, our experiments revealed that index-masking [113], a well-known defense used against Spectre-PHT in WebKit for example, *may introduce new Spectre-STL vulnerabilities*. We propose and verify safe implementations to deal with this problem. By means of our tool, we have also discovered that a popular option [112] of `gcc` to generate position-independent code (PIC) *may introduce Spectre-STL vulnerabilities*. We also confirm, as already reported by Cauligi et al. [67], that the stack protections added by compilers introduce Spectre violations in cryptographic primitives.

**Contributions.** In summary, our contributions are:

— We design a dedicated technique on top of relational symbolic execution, named *Haunted RelSE*, to efficiently analyze speculative executions in symbolic analysis to detect PHT and STL Spectre violations (Sections 6.2 and 6.3). The main idea behind Haunted RelSE is to *symbolically reason on sequential and transient behaviours at the same time*. Even though our encoding for memory speculations is reminiscent of some encodings for state merging [135, 120, 160], we actually follow a different philosophy, by preventing artificial splits between sequential and transient executions rather than trying to pack together different (possibly unrelated) paths. We formally prove that Haunted RelSE is semantically equivalent to a relational analysis modeling all speculative executions explicitly (Section 6.3);

— We propose a verification tool, called Binsec/Haunted, implementing Haunted RelSE (Section 6.4) and evaluate it on well-known litmus tests (small test cases) for Spectre-PHT. We further propose a new set of litmus tests for Spectre-STL as a contribution and test Binsec/Haunted on it. Experimental evaluation (Section 6.5) shows that Binsec/Haunted can find violations of speculative constant-time in real-world cryptographic code, such as donna, Libsodium and OpenSSL libraries. For Spectre-PHT, Binsec/Haunted can exhaustively analyze code up to 5k static instructions. It is faster than the (less precise) state of the art tools KLEESpectre and Pitchfork. For Spectre-STL, it can exhaustively analyze code up to 100 instructions and find vulnerabilities in code up to 6k instructions; compared to the state-of-the art tool Pitchfork,

BINSEC/HAUNTED is significantly faster, finds more vulnerabilities, and report more insecure programs;

— To the best of our knowledge, we are the first to report that the well-known defense against Spectre-PHT, index-masking, may introduce Spectre-STL vulnerabilities. We propose correct implementations, verified with our tool, to remedy this problem (Section 6.6.1). We are also the first to report that PIC options [112] from the gcc compiler introduce Spectre-STL violations (Section 6.6.2).

**Discussion.** While Spectre attacks opened a new battlefield of system security, reasoning about speculative executions is hard and tedious. There is a need for automated analysis techniques, yet prior proposals suffer from scalability issues due to the path explosion induced by extra speculative behaviors. Haunted RelSE allows to prune part of this complexity, making a step towards scalable analysis of Spectre attacks. For Spectre-PHT, Haunted RelSE can dramatically speed up the analysis in some cases, pruning the complexity of analyzing speculative semantics, and scales on medium-size real-world cryptographic binaries. For Spectre-STL, it is the first tool able to exhaustively analyze small real world cryptographic binaries and find vulnerabilities in medium-size real world cryptographic binaries.

**Related background**

The following background is recommended before reading this chapter:

— introduction of symbolic execution, given in Section 2.2,

— introduction of transient execution attacks definition of speculative-time, given in Section 3.3,

— the basics of binary analysis, given in Section 2.3—in particular the low-level language used in this section, defined in Section 2.3.4 and the binary-level symbolic execution on which we build, defined in Section 2.3.5.

Moreover, in order for this chapter to be self contained, we recall in Section 6.2.1 the key concepts and notations of binary-level relational symbolic execution (RelSE) presented in Section 4.4.1.

## 6.2 Haunted RelSE in a nutshell

**Section overview**

This section gives an overview of how to modify RelSE to consider the speculative semantics of the program [67] to model Spectre-PHT (cf. Section 6.2.2) and to model Spectre-STL (cf. Section 6.2.3). Before that it first recall the key concepts and notations of RelSE (cf. Section 6.2.1).

The speculative semantics includes:

— *Sequential executions*: instructions that are executed as a result of a good speculation and are kept once the speculation is resolved;

— *Transient executions*: instructions that are executed as a result of a misspeculation and that are discarded once the speculation is resolved.

Two solutions to the problem are presented:

1. The approach that models transient executions explicitly—employed in state-of-the art tools (cf. Table 6.6)—which we call *Explicit*;
2. Our optimized exploration strategy, which we call *Haunted*.

### 6.2.1 Binary-level RelSE in a nutshell

**Reminder (cf. Section 2.3.5)**

Binary-level symbolic execution relies on the quantifier-free theory of fixed-size bitvectors and arrays (QF_ABV [29]). We let $\Phi$ denote the set of symbolic expressions in the QF_ABV logic and $\varphi$, $\phi$, $\psi$, $\iota$ be symbolic expressions ranging over $\Phi$. In particular, $\mathcal{B}v_n \subseteq \Phi$ and $\mathcal{B}l \subseteq \Phi$ respectively denote the set of symbolic $n$-bit bitvectors and boolean expressions.

A model $M$ assigns concrete values to symbolic variables. The satisfiability of a formula $\pi$ with a model $M$ is denoted $M \vDash \pi$. In the implementation, an SMT solver is used to determine satisfiability of a formula and obtain a satisfying model, denoted $M \vDash_{\text{SMT}} \pi$. Whenever the model is not needed for our purposes, we leave it implicit and simply write $\vDash \pi$ or $\vDash_{\text{SMT}} \pi$ for satisfiability.

**Reminder (cf. Section 4.4.1)**

A *relational* expression $\widehat{\varphi}$ is either a simple symbolic expression $\langle\varphi\rangle$ or a pair $\langle\varphi_l \,|\, \varphi_r\rangle$ of two symbolic expression in $\Phi$. We denote $\widehat{\varphi}_{|l}$ (resp. $\widehat{\varphi}_{|r}$), the projection on the left (resp. right) value of $\widehat{\varphi}$. If $\widehat{\varphi} = \langle\varphi\rangle$, then $\widehat{\varphi}_{|l}$ and $\widehat{\varphi}_{|r}$ are both defined as $\varphi$. Let $\boldsymbol{\Phi}$ be the set of relational formulas and $\boldsymbol{\mathcal{B}v_n}$ be the set of relational symbolic bitvectors of size $n$.

**Reminder (cf. Section 4.4.1.1)**

For the security evaluation, we define a predicate $secLeak : \boldsymbol{\Phi} \times \Phi \to Bool$, which ensures that a relational formula does not differ in its right and left components, meaning that it can be leaked securely:

$$secLeak(\widehat{\varphi}, \pi) = \begin{cases} true & \text{if } \widehat{\varphi} = \langle\varphi\rangle \\ true & \text{if } \widehat{\varphi} = \langle\varphi_l \,|\, \varphi_r\rangle \wedge \nvDash_{\text{SMT}} \pi \wedge \varphi_l \neq \varphi_r \\ false & \text{otherwise} \end{cases}$$

### 6.2.2 Spectre-PHT

This section presents how to adapt RelSE to model Spectre-PHT attacks. It starts by introducing the classical *Explicit* solution in Section 6.2.2.1 and presents our optimized exploration strategy, *Haunted*, in Section 6.2.2.2.

#### 6.2.2.1 Explicit RelSE for Spectre-PHT

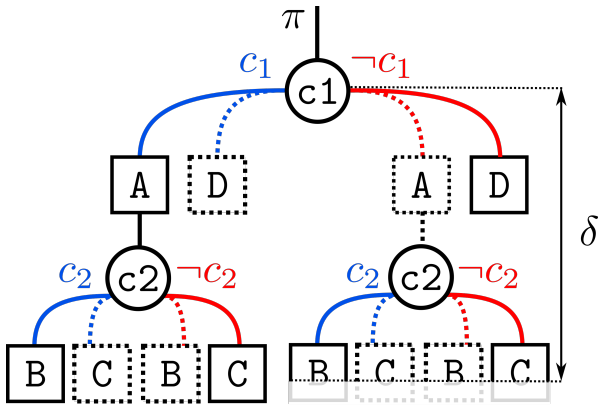
The *Explicit* approach to model Spectre-PHT in symbolic execution—introduced in KLEESpectre [252]—explicitly models transient executions by forking into four paths at each conditional branch. Consider for instance, the program in Figure 6.1a and its symbolic execution tree in Figure 6.1b. After the conditional instruction `if c1` the execution forks into four paths:

— Two *sequential paths*: Like in standard symbolic execution, the first path follows the *then* branch and adds the constraint $(c_1 = true)$ to the path predicate; whereas the second path follows the *else* branch with the constraint $(c_1 = false)$;

— Two *transient paths*: To account for transient executions that are mispredicted to *true*, the *then* branch is executed with the constraint $(c_1 = false)$; whereas to account for transient executions that are mispredicted to $false$, the *else* branch is executed with the constraint $(c_1 = true)$. These transient paths are discarded after reaching a *speculation bound* (usually defined by the size of the reorder buffer).

```
int main() {
  if (c1) {
    A()
    if (c2) {
      B()
    } else {
      C()
  } else {
    D()
}}}
```

(A) Illustrative program.




(B) Explicit RelSE.

FIGURE 6.1 – Example of RelSE for speculative semantics with the *Explicit* exploration strategy where solid paths represent sequential executions, dotted paths represent transient executions, and $\delta$ is the speculation depth.

Finally, to verify speculative constant-time, we have to check that memory accesses and conditional statements do not leak secret information on both sequential paths and transient paths:

— On sequential paths, we check that the *control-flow* of the program and the *indexes of load and store* instructions do not depend on the secret input;

— However, on transient paths, we only check the *control-flow* and the *index of load* instructions. Reason is that, in speculative execution, memory stores are queued in the store buffer and are invisible to the cache until they are retired [252].

**Problem with Explicit** In Figure 6.1b, we can see that both subtrees resulting from executing the *then* branch in sequential and transient execution (i.e. subtrees starting from state `A`) correspond to the same instructions under different path predicates. Precisely, if we call $\widehat{\psi}_{cf}$, $\widehat{\psi}_{ld}$, and $\widehat{\psi}_{st}$ the relational expressions corresponding respectively to control-flow statements, load indexes and store indexes in subtree `A`, then we have to check the following equation for the sequential execution:

$$secleak(\pi \wedge c_1, \widehat{\psi}_{cf}) \wedge secleak(\pi \wedge c_1, \widehat{\psi}_{ld}) \wedge secleak(\pi \wedge c_1, \widehat{\psi}_{st})$$

and a very similar equation for the transient execution:

$$secleak(\pi \wedge \neg c_1, \widehat{\psi}_{cf}) \wedge secleak(\pi \wedge \neg c_1, \widehat{\psi}_{ld})$$

In the end, this is equivalent to checking the formula:

$$secleak(\pi, \widehat{\psi}_{cf}) \wedge secleak(\pi, \widehat{\psi}_{ld}) \wedge secleak(\pi \wedge c_1, \widehat{\psi}_{st})$$

This formula essentially amounts to symbolically executing the *then* branch up to $\delta$, checking load indexes $\widehat{\psi}_{ld}$ and control-flow expressions $\widehat{\psi}_{cf}$ without adding the constraint $c_1$, and only add $c_1$ to check store indexes $\widehat{\psi}_{st}$.

*This observation led us to design an optimization of Explicit RelSE: we can explore a single* speculative path *that encompasses both the* sequential *and the* transient *paths after a conditional statement, in order to prune states while keeping an equivalent result.*

#### 6.2.2.2 Haunted RelSE for Spectre-PHT

Instead of forking the execution into four paths, *Haunted RelSE* forks the execution into two paths, as illustrated in Figure 6.2. After the conditional branch `if c1`, the execution forks into two paths:

— a path following the *then* branch (subtree `A`),

— a path following the *else* branch (subtree `D`).

Both paths model the behavior of the sequential and the corresponding transient paths *at the same time*, under the path predicate $\pi$. Moreover, it delays satisfiability checks of the path predicate—and possibly spares them if the path is terminated before conditions are retired. Finally, the constraint is added to the path predicate when the conditional branch is retired (after $\delta$ steps).

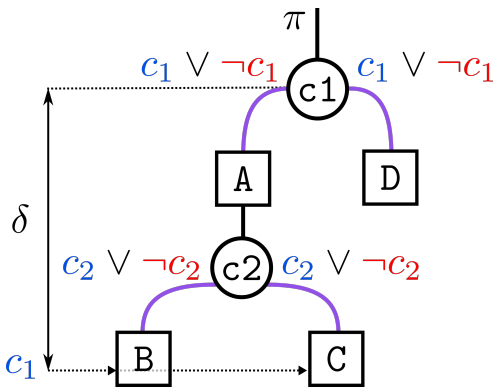



FIGURE 6.2 – Haunted RelSE of program in Figure 6.1a where $\delta$ is the speculation depth. The constraint $c_1 \vee \neg c_1$ is added only for clarity.

At each conditional statement (resp. load instruction), we check that the condition (resp. load index) does not depend on the secret in both the *sequential and transient* executions, using the path predicate $\pi$:

$$secleak(\pi, \widehat{\psi}_{cf}) \wedge secleak(\pi, \widehat{\psi}_{ld})$$

On the other hand, store instructions are checked under the *sequential execution only*, using the path predicate $\pi \wedge c_1$:

$$secleak(\pi \wedge c_1, \widehat{\psi}_{st})$$

Finally, the condition $(c_1 = true)$ is added to the path predicate after $\delta$ steps.

**Performance.** In the best case—when both $\pi \wedge c_1$ and $\pi \wedge \neg c_1$ are satisfiable—*Haunted RelSE* completely prunes the complexity of analyzing transient executions, making it equivalent to standard RelSE; whereas *Explicit RelSE* has a quadratic explosion of the search space. However, in the worst case—when only one path is satisfiable—*Haunted RelSE* and *Explicit RelSE* are equivalent.

Our experimental evaluation in Section 6.5.2, confirms this theoretical performance. In most cases, *Haunted RelSE* improves performance over *Explicit RelSE*. In the worse case, it is equivalent to *Explicit RelSE*; whereas in the best case, it gives a speedup of 3 orders of magnitude.

### 6.2.3 Spectre-STL

This section presents how to adapt RelSE to model Spectre-STL attacks. It starts by introducing the classical *Explicit* solution in Section 6.2.3.1 and presents our optimized exploration strategy, *Haunted*, in Section 6.2.3.2.

#### 6.2.3.1 Explicit RelSE for Spectre-STL

At the microarchitectural level, a load instruction can take its value from any matching entry in the store buffer; or from the main memory. In other words, the load can bypass each pending store in the store buffer until it reaches the main memory. To account for this behavior, the *Explicit* strategy—employed in PITCHFORK [67]—is to fork the symbolic execution for each possible load and store interleaving [2].

**Symbolic memory.** Consider as an illustration the program in Figure 6.3a. Symbolic execution of the store instructions (block `S`) produces the symbolic memory $\mu_3$ defined in Figure 6.3b, which is the sequence of symbolic store operations starting from $initial\_memory$. With this chronological representation, we can easily define the content of a *store buffer* of size $|SB|$ by taking the $|SB|$ last store operations of the symbolic memory. Similarly, the *main memory* can be defined by removing the last $|SB|$ store operations from the symbolic memory. If we consider a store buffer of size 2, the last two store expressions constitute the store buffer whereas the main memory is defined by $\mu_1$.

**Evaluation of loads.** The first load instruction (block `A`) can bypass each store operation in the store buffer until it reaches the main memory. Therefore there are three possible values for $x$, as detailed in Figure 6.3c:

— The *sequential value* $s$ corresponds to a symbolic *select* operation from the most recent symbolic memory $\mu_3$. Because all prior store operations are encoded in-order into $\mu_3$, this corresponds to the in-order execution;

— The first *transient value* $t_2$ is obtained by bypassing the first entry in the store buffer. This corresponds to a symbolic *select* operation from $\mu_2$;

— The final *transient value* $t_1$ is obtained by bypassing the first and the second entries in the store buffer and taking its value from the main memory. This corresponds to a symbolic *select* operation from $\mu_1$.

Similarly, variable $y$ also has three possible values.

To model these multiple choices in symbolic execution, the *Explicit* exploration strategy forks for each possible value that a load can take, as illustrated in Figure 6.3d.

2. In a window of 20 instructions in PITCHFORK.

```
store a1 v1;   } S
store a2 v2;
store a3 v3;
x = load a;    } A
y = load b;    } B
[...]          } C
```

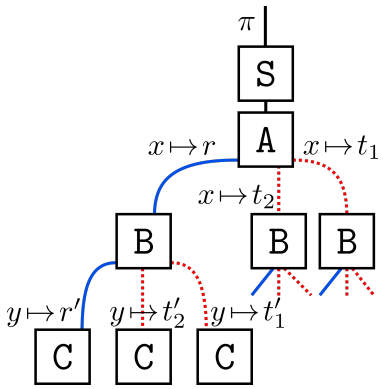
(A) Illustrative program.

$$\left.\begin{array}{l}\mu_0 = initial\_memory\\ \mu_1 = store\ \mu_0\ a_1\ v_1\end{array}\right\}\text{Mem}\qquad \left.\begin{array}{l}\mu_2 = store\ \mu_1\ a_2\ v_2\\ \mu_3 = store\ \mu_2\ a_3\ v_3\end{array}\right\}\text{SB}$$

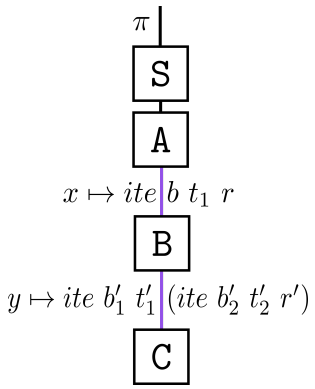
(B) Symbolic memory where SB is the store buffer (of size 2) and Mem is the main memory.

$$x = \left\{\begin{array}{l} s \triangleq select\ \mu_3\ a\\ t_2 \triangleq select\ \mu_2\ a\\ t_1 \triangleq select\ \mu_1\ a\end{array}\right\} \quad y = \left\{\begin{array}{l} s' \triangleq select\ \mu_3\ a'\\ t'_2 \triangleq select\ \mu_2\ a'\\ t'_1 \triangleq select\ \mu_1\ a'\end{array}\right\} \quad \begin{array}{l}\text{In-order execution}\\ \text{Bypass } 1^{\text{st}} \text{ SB entry}\\ \text{Bypass } 1^{\text{st}} \text{ \& } 2^{\text{nd}} \text{ SB entries}\end{array}$$

(C) Symbolic evaluation of loads.




(D) Explicit RelSE.




(E) Haunted RelSE when $a \neq a_2$.

FIGURE 6.3 – Symbolic evaluation, under a speculative semantics, of the program in Figure 6.3a. The symbolic memory is given in Figure 6.3b and the symbolic evaluation of load instructions is detailed in Figure 6.3c. Figure 6.3d illustrates the symbolic execution tree obtained from the Explicit exploration strategy, where solid paths denote sequential executions and dotted paths denote transient executions; and Figure 6.3e, the tree obtained from Haunted RelSE.

Unfortunately, this quickly leads to path explosion and we show experimentally (Section 6.5.3) that this solution is intractable even on small codes (100 instructions).

#### 6.2.3.2 Haunted RelSE for Spectre-STL

**Pruning redundant paths.** The first observation that we make is that most paths are redundant as a load can naturally commute with non-aliasing prior stores. Take, for instance, the evaluation of loads in Figure 6.3c. If we can determine that the index of the load, $a$, is distinct from the index of the second store, $a_2$, then by the theory of arrays, we have $select\ \mu_2\ a = select\ \mu_1\ a$. Therefore $t_2 = t_1$, meaning that the path $x \mapsto t_2$ in Figure 6.3d and all of its subpaths are redundant. *We rely on a well-known optimization for symbolic arrays called read-over-write [107] to detect and prune these redundant cases.*

**Encoding remaining cases.** Merely pruning redundant cases is not sufficient to deal with path explosion (as we show in Section 6.5.3), thus *we propose a dedicated encoding to keep the remaining cases in a single path.* We use symbolic *if-then-else* to encode in a single expression all the possible values that a load can take instead of forking the execution for each possible case, as illustrated in Figure 6.3e.

Take, for instance, the evaluation of load expressions given in Figure 6.3c where $y$ can take the values $s'$, $t'_1$, or $t'_2$. We introduce two fresh symbolic boolean variables $b'_1$ and $b'_2$ and build the expression $(ite\ b'_1\ t'_1\ (ite\ b'_2\ t'_2\ r'))$. The solver can let $y$ take the following values:

— transient value $t'_1$ by setting $b'_1$ to true,

— transient value $t'_2$ by setting $b'_1$ to false and $b'_2$ to true,

— sequential value $s'$ by setting both $b'_1$ and $b'_2$ to $false$.

Finally, transient values $t'_1$ (resp. $t'_2$) can be easily discarded (e.g. after reaching the speculation depth) by setting $b'_1$ (resp. $b'_2$) to false.

## 6.3 Formalization of Haunted RelSE

**Section overview**

This section introduces the technical details of *Haunted RelSE*:

— First, it introduces *dated expressions* and details the symbolic evaluation of expressions (cf Section 6.3.1);

— Next, it presents the technical details of Haunted RelSE for Spectre-PHT (cf. Section 6.3.2), and for Spectre-STL (cf. Section 6.3.3);

— Then, it presents the rules for the symbolic evaluation of expressions in Section 6.3.4, wrapping-up the details presented in Sections 6.3.2 and 6.3.3;

— Finally, it shows that *Haunted RelSE* is semantically equivalent to *Explicit RelSE* (cf. Section 6.3.5).

Notice that, most instructions naturally commute or cannot be reordered because of their data dependencies. Consequently, we only need to consider reordering of conditional branches for Spectre-PHT and reordering of `load` and `store` instructions for Spectre-STL.

### 6.3.1 Symbolic evaluation of expressions

**Dated expression.** Instead of modeling the reorder buffer explicitly, we use the *current depth* of the symbolic execution, denoted $\widetilde{\delta}$, to track expressions to retire. Conditions and load operations that are speculatively executed, are associated with a *retirement depth*, denoted $(\widehat{\varphi}, \delta)$, meaning that expression $\widehat{\varphi}$ must be retired when $\delta \leq \widetilde{\delta}$.

Additionally, expressions computed during symbolic execution are annotated with *memory-dependency depth*, denoted $e^{\delta}$, to determine whether they depend on the memory. Consequently, the register map $\rho$, maps a variable $v$ to a dated expression $\widehat{\varphi}^{\delta}$; intuitively, $\delta$ is the retirement depth of the last memory access on which $v$ depends. In other words, we can consider that an expression $\widehat{\varphi}^{\delta}$ does not depend on the memory (i.e. can be computed quickly, from registers) when the symbolic configuration has reached depth $\delta$. The memory-dependency depth is used to determine the speculation

depth of conditional instruction in Section 6.3.2.1. When $\delta$ is not needed in the context, it is omitted.

A summary on how to interpret dated expressions is given in Table 6.1.

| | |
|---|---|
| $\widetilde{\delta}$ | *Current depth* of the symbolic execution. |
| $\Delta$ | Maximum size of the reorder buffer (given by the microarchitecture); It determines the *maximum speculation depth*. |
| $(\widehat{\varphi}, \delta)$ | Expression $\widehat{\varphi}$ with a *retirement depth* $\delta$; $\widehat{\varphi}$ will be retired when $\delta \leq \widetilde{\delta}$. |
| $\widehat{\varphi}^{\delta}$ | Expression $\widehat{\varphi}$ with a *memory-dependency depth* $\delta$; $\widehat{\varphi}$ depends on memory until $\delta \leq \widetilde{\delta}$. |

TABLE 6.1 – Interpretation of dated expressions.

**Symbolic configuration.** Let $\wp(S)$ denote the powerset of $S$. A *symbolic configuration* is of the form $\left(l, \widetilde{\delta}, \rho, \widehat{\mu}, \pi, \widetilde{\pi}, \widetilde{\lambda}\right)$ where:

- $l \in \mathit{Loc}$ is the current program point, which is used to get the current instruction in the program $\mathbb{P}$, denoted $\mathbb{P}[l]$;
- $\widetilde{\delta} \in \mathbb{N}$ is the current depth of the symbolic execution;
- $\rho : \mathcal{V} \to \mathbf{\Phi} \times \mathbb{N}$ is a symbolic register map, mapping variables from $\mathcal{V}$ to their symbolic representation as a relational formula in $\mathbf{\Phi}$, along with their *memory-dependency depth*;
- $\widehat{\mu} : (\mathit{Array}\ \mathcal{B}v_{32}\ \mathcal{B}v_8) \times (\mathit{Array}\ \mathcal{B}v_{32}\ \mathcal{B}v_8) \times \mathbb{N}$ is the symbolic memory—a pair of arrays of values in $\mathcal{B}v_8$ indexed by addresses in $\mathcal{B}v_{32}$, along with the *retirement depth* of their store operations;
- $\pi \in \Phi$ is the *retired path predicate*—a conjunction of constraints recording retired branch conditions (details in Section 6.3.2);
- $\widetilde{\pi} : \wp(\mathcal{B}l \times \mathbb{N})$ is the *speculative path predicate*—a set of speculatively executed branch conditions and their *retirement depth* (details in Section 6.3.2);
- $\widetilde{\lambda} : \wp(\mathcal{B}l \times \mathbb{N})$ is the *transient load set*—a set of boolean variables corresponding to transient load values with their *retirement depth* (details in Section 6.3.3).

The notation $s.f$ is used to denote the field $f$ in configuration $s$. We also define a function $\mathit{eval\_expr}(s, e)$ which evaluates a DBA expression $e$ to a symbolic value in a symbolic configuration $s$.

**Evaluation of expressions.** Symbolic evaluation of a DBA expression $e$ in a configuration $s$ evaluates to a relational expression $\widehat{\varphi}$, with a memory-dependency depth $\delta$, and the updated set of transient loads $\widetilde{\lambda}$. It is denoted $s\ e \vdash \widehat{\varphi}^{\delta}, \widetilde{\lambda}$ and detailed in Figure 6.4. Detailed explanations of the rules follow:

- CST is the evaluation of a constant `bv` and returns the corresponding symbolic bitvector $bv$ as a simple expression, with a depth set to $-\infty$ (i.e. the expression does not depend on the memory);
- VAR is the evaluation of a variable `v` and returns the mapping of `v` from the register map $\rho$;
- UNOP is the evaluation of a unary operator $\blacktriangleleft e$. First, it evaluates the expression $e$ to a dated expression $\widehat{\phi}^{\delta}$. Then it computes the value $\widehat{\varphi}$ by applying the symbolic operator $\blacktriangleleft$ to $\widehat{\phi}$. Finally, it returns the new value $\widehat{\varphi}$ with depth $\delta$;

— BINOP is the evaluation of binary operators—similar to the rule UNOP. The memory-dependency depth of the result is the maximum of the memory-dependency depth of its operands, meaning that the expression cease to depend on the memory when *all* the expressions on which it depends cease to depend on the memory;

— LOAD is the evaluation of load expressions (also detailed as an algorithm in Algorithm 7). The first constraint is to verify that the leakage of the index is secure (i.e. $secLeak(\hat{\imath}, \pi)$). Then the rule calls a lookup function $lookup_{ite}$, which returns the set of symbolic values that the load can take, encoded as a single if-then-else expression $\widehat{\varphi}$, and updates the set of transient load $\widetilde{\lambda}$ (details in Section 6.3.3.2). The rule returns the symbolic expression $\widehat{\varphi}$, marked with the retirement depth of the instruction ($\widetilde{\delta} + \Delta$). This depth is the *memory dependency depth* of the expression and will later be used when determining the speculation depth of conditional instructions (details in Section 6.3.2.1).

**Expr**

$$\text{CST}\ \frac{}{(\_, \widetilde{\lambda})\ \texttt{bv} \vdash \langle bv \rangle^{-\infty}, \widetilde{\lambda}} \qquad \text{VAR}\ \frac{}{(\_, \rho, \_)\ \texttt{v} \vdash \rho\ \texttt{v}, \widetilde{\lambda}}$$

$$\text{UNOP}\ \frac{(\_)\ e \vdash \widehat{\phi}^{\delta}, \widetilde{\lambda}_e \qquad \widehat{\varphi} \triangleq \blacktriangleleft\, \widehat{\phi}}{(\_)\ \blacktriangleleft\, e \vdash \widehat{\varphi}^{\delta}, \widetilde{\lambda}_e}$$

$$\text{BINOP}\ \frac{\begin{array}{c}(\_, \widetilde{\lambda})\ e_1 \vdash \widehat{\phi}^{\delta_1}, \widetilde{\lambda}_1 \\ (\_, \widetilde{\lambda}_1)\ e_2 \vdash \widehat{\psi}^{\delta_2}, \widetilde{\lambda}_2 \qquad \widehat{\varphi} \triangleq \widehat{\phi} \blacklozenge \widehat{\psi} \qquad \boxed{\delta \triangleq max(\delta_1, \delta_2)}\end{array}}{(\_, \widetilde{\lambda})\ e_1 \diamond e_2 \vdash \widehat{\varphi}^{\delta}, \widetilde{\lambda}_2}$$

$$\text{LOAD}\ \frac{\begin{array}{c}(\_, \widetilde{\delta}, \_, \widehat{\mu}, \pi, \_)\ e \vdash \hat{\imath}, \widetilde{\lambda}' \\ \boxed{\widehat{\varphi}, \widetilde{\lambda}'' \triangleq lookup_{ite}(\widetilde{\lambda}', \widehat{\mu}, \hat{\imath}, \widetilde{\delta})} \qquad secLeak(\hat{\imath}, \pi)\end{array}}{(\_, \widetilde{\delta}, \_, \widehat{\mu}, \pi, \_)\ \texttt{load}\ e \vdash \boxed{\widehat{\varphi}^{\widetilde{\delta}+\Delta}}, \widetilde{\lambda}''}$$

FIGURE 6.4 – Relational symbolic evaluation of DBA expressions using Haunted RelSE, where $\blacktriangleleft$ (resp. $\diamond$) are the symbolic counterpart of concrete operators $\blacktriangleleft$ (resp. $\blacklozenge$). For readability, terms that are not important in the context are replaced with _ and important details are highlighted with *boxes*.

### 6.3.2 Haunted RelSE for Spectre-PHT

**Section overview**

This section, formalizes *Haunted RelSE* for Spectre-PHT. First, it presents a dynamic approach to determine the speculation depth, in which a condition is retired as soon as memory accesses on which it depends are resolved (cf. Section 6.3.2.1). Then, it introduces the evaluation of conditional instructions (cf. Section 6.3.2.2). Finally, it details how to invalidate transient paths when their speculation depth has been reached (cf. Section 6.3.2.3).

#### 6.3.2.1 Dynamic speculation depth

The *retirement depth* of conditional branches is computed dynamically, considering that a condition can be fully resolved (and mispredicted paths can be squashed) when all the memory accesses on which it depends are retired. In particular it means that if the condition does not depend on the memory then the branch is not mispredicted [261, 132]. This requires to keep, for each expression, the retirement depth of its last memory dependency (i.e. its *memory dependency depth*).

In particular, at a conditional branch $\texttt{ite c ? } l_{true} : l_{false}$, c evaluates to a dated expression $\widehat{\varphi}^{\delta}$, where $\delta$ is its memory-dependency depth (cf. LOAD rule in Section 6.3.1). This depth $\delta$ is added to the speculative path predicate $\widetilde{\pi}$ as the *retirement depth* of the condition.

#### 6.3.2.2 Evaluation of conditional instructions

Contrary to standard symbolic execution, conditions are not added to the path predicate right away. Instead, they are kept in a *speculative path predicate*, denoted $\widetilde{\pi}$, along with their retirement depth. When the retirement depth of a condition is reached, it is removed from the speculative path predicate and added to the *retired path predicate*, denoted $\pi$.

Evaluation of conditional branches is detailed in Algorithm 2. First, the function evaluates the symbolic value of the condition and checks that it can be leaked securely. Then it computes the two next states $s_t$, following the *then* branch, and $s_f$, following the *else* branch by updating the location and the speculative path predicate $\widetilde{\pi}$.

**Func** *eval_ite*$(s)$ **where** $\mathbb{P}[s.l] = \texttt{ite c ? } l_{true} : l_{false}$ **is**
  **require** $\mathbb{P}[s.l] = \texttt{ite c ? } l_{true} : l_{false}$;
  $s\ c \vdash \widehat{\varphi}^{\delta}$; ▷ *Evaluates condition c to* $\widehat{\varphi}^{\delta}$
  **assert** *secLeak*$(\widehat{\varphi}, s.\pi)$; ▷ *Ensure that leakage of c is secure*
  ▷ *Compute state following the then branch*
  $s_t \leftarrow s$;
  $s_t.l \leftarrow l_t$;
  $s_t.\widetilde{\pi} \leftarrow s.\widetilde{\pi} \cup \{(\widehat{\varphi}, \delta)\}$;
  ▷ *Compute state following the else branch*
  $s_f \leftarrow s$;
  $s_f.l \leftarrow l_f$;
  $s_f.\widetilde{\pi} \leftarrow s.\widetilde{\pi} \cup \{(\neg\widehat{\varphi}, \delta)\}$;
  **return** $(s_t, s_f)$

**Algorithm 2:** Evaluation of conditional branches. The depth $\delta$ added to the speculative path predicate $\widetilde{\pi}$ is the retirement depth of the condition. Once the symbolic execution reaches this retirement depth (i.e. $s.\widetilde{\delta} \leq \delta$), the condition is added to the retired path predicate $\pi$ and we are no longer in speculative execution (see Section 6.3.2.3).

#### 6.3.2.3 Invalidate transient paths

Conditional branches are retired in the function $retire_{\text{PHT}}(\pi, \widetilde{\pi}, \widetilde{\delta})$ defined in Algorithm 3. The function removes from the speculative path predicate $\widetilde{\pi}$ all the conditions with retirement depth $\delta$ below the current depth $\widetilde{\delta}$, and adds them to the retired path

predicate $\pi$. It returns the updated path predicates $\pi$ and $\widetilde{\pi}$. The symbolic execution stops, if $\pi$ becomes unsatisfiable.

**Func** *retire*PHT$(\pi, \widetilde{\pi}, \widetilde{\delta})$ **is**
  $\pi' \leftarrow \pi$;
  $\widetilde{\pi}' \leftarrow \emptyset$;
  **for** $(\widehat{\varphi}, \delta)$ **in** $\widetilde{\pi}$ **do**
    **if** $\delta \leq \widetilde{\delta}$ **then**
      $\pi' \leftarrow \pi' \wedge \widehat{\varphi}$; ▷ *Retire the condition*
    **else**
      $\widetilde{\pi}' \leftarrow \widetilde{\pi}' \cup \{(\widehat{\varphi}, \delta)\}$
  **return** $(\pi', \widetilde{\pi}')$

**Algorithm 3:** Retire expired conditions.

### 6.3.3 Haunted RelSE for Spectre-STL

**Section overview**

This section, formalizes *Haunted RelSE* for Spectre-STL. First, it defines the symbolic memory and the evaluation of `store` instructions (cf. Section 6.3.3.1). Then it introduces the evaluation of `load` instructions, accounting for transient load values (cf. Section 6.3.3.2). Finally it details how to invalidate transient load values when their speculation depth has been reached (cf. Section 6.3.3.3).

#### 6.3.3.1 Symbolic memory

In a symbolic configuration, the memory $\widehat{\mu}$ is the history of symbolic *store* operations starting from the initial memory (as in Section 4.4). In addition, to model the speculative semantics, each symbolic *store* is annotated with a *retirement depth*, which indicates when the store is still in the *store buffer* (see Definition 15) or when it is retired, i.e. committed to the *main memory* (see Definition 16). The retirement depth of the initial symbolic memory is set to $-\infty$ (i.e. we consider that memory initialization cannot be bypassed).

**Evaluation of store instructions.** The evaluation of `store` instructions is detailed in Algorithm 4. First, the function evaluates the symbolic values of the index and ensures that it can be leaked securely under the *sequential path predicate* $\pi_{seq}$. The sequential path predicate is the conjunction of the retired path predicate $\pi$ with all the pending conditions in $\widetilde{\pi}$, plus the invalidation of the transient loads in $\widetilde{\lambda}$ (detailed in Section 6.3.3.3). Then, it updates the symbolic memory with a symbolic *store* operation and sets the retirement depth of this store to $\widetilde{\delta} + \Delta$ (i.e. the current depth plus the maximum speculation depth).

**Store buffer.** The *store buffer*, is the restriction of the symbolic memory to the last $|SB|$ store operations which have not been retired—where $|SB|$ is the size of the store buffer, defined by the microarchitecture:

**Definition 15** (Store buffer $(SB(\widehat{\mu}, \widetilde{\delta}))$)**.** *The content of a store buffer of size* $|SB|$*, for a symbolic memory* $\widehat{\mu}$ *and a current depth* $\widetilde{\delta}$ *is defined as:*

$$SB(\widehat{\mu}, \widetilde{\delta}) \triangleq \{(s, \delta) \mid (s, \delta) \in last(|SB|, \widehat{\mu}) \wedge \delta > \widetilde{\delta}\}$$

**Func** *eval_store*$(s)$ **where** $\mathbb{P}[s.l] =$ `store` $i$ $v$ **is**
- $s\ i \vdash \widehat{\imath}$; ▷ *Evaluates index* $i$ *to* $\widehat{\imath}$
- $s\ v \vdash \widehat{\nu}$; ▷ *Evaluates value* $v$ *to* $\widehat{\nu}$
- $\pi_{seq} \triangleq \mathit{retire}\textsc{All}(s.\pi, s.\widetilde{\pi}, s.\widetilde{\lambda}, s.\widetilde{\delta})$;
- **assert** $\mathit{secLeak}(\widehat{\imath}, \pi_{seq})$; ▷ *Ensure that leakage of* $\widehat{\imath}$ *is secure*
- $s' \leftarrow s$;
- $s'.l \leftarrow s.l + 1$; ▷ *Go to next location*
- ▷ *Update memory and set retirement depth in* $\Delta$ *steps*
- $s'.\widehat{\mu} \leftarrow (\boldsymbol{store}(s.\widehat{\mu}, \widehat{\imath}, \widehat{\nu}), \widetilde{\delta} + \Delta)$;
- **return** $s'$

**Algorithm 4:** Evaluation of store instructions where *retire*ALL returns the sequential path predicate (details in Section 6.3.3.3).

*where* $\mathit{last}(n, \widehat{\mu})$ *is the projection on the last* $n$ *element of the symbolic memory* $\widehat{\mu}$.

**Main memory.** Conversely, the *main memory* is defined as the restriction of the symbolic memory to the retired store operations:

**Definition 16** (Main memory ($\mathit{Mem}(\widehat{\mu}, \widetilde{\delta})$))**.** *The content of the main memory, for a symbolic memory* $\widehat{\mu}$ *and a current depth* $\widetilde{\delta}$ *is defined as:*

$$\mathit{Mem}(\widehat{\mu}, \widetilde{\delta}) \triangleq \widehat{\mu} \setminus \mathit{SB}(\widehat{\mu}, \widetilde{\delta})$$

#### 6.3.3.2 Evaluation of load expressions

Load expressions can either take their value from a pending store in the store buffer with a matching address via *store-to-load forwarding*; or can speculatively bypass pending stores in the store buffer and take their value from the main memory [144]. Instead of considering all possible interleavings between a load expression and prior stores in the store-buffer, we use *read-over-write* [107] to identify and discard most cases in which the load and a prior store naturally commute. *Read-over-write* is a well known simplification for the theory of arrays which resolves select operations on symbolic arrays ahead of the solver.

**Reminder (cf. Section 4.4.2.1)**

**Read-over-write.** To resolve select operations ahead of the solver, read-over-write defines a $\mathit{lookup}_{mem}$ function:

$$\mathit{lookup}_{mem}(\widehat{\mu}_0, i) \triangleq \boldsymbol{select}(\widehat{\mu}_0, i)$$

$$\mathit{lookup}_{mem}(\widehat{\mu}_n, i) \triangleq \begin{cases} \widehat{\varphi} & \text{if } eq^{\#}(i,j) \\ \mathit{lookup}_{mem}(\widehat{\mu}_{n-1}, i) & \text{if } \neg eq^{\#}(i,j) \\ \boldsymbol{select}(\widehat{\mu}_n, i) & \text{if } eq^{\#}(i,j) = \bot \end{cases}$$

$$\text{where } \widehat{\mu}_n \triangleq \boldsymbol{store}(\widehat{\mu}_{n-1}, j, \widehat{\varphi})$$

where $eq^{\#}(i,j)$ is a comparison function, relying on *syntactic term equality* to efficiently compare indexes. It returns true (resp. false) only if $i$ and $j$ are syntactically equal (resp. different). If the terms are not comparable, it is undefined, denoted $\bot$.

**Lookup in store-buffer.** In order to model store-to-load forwarding efficiently, we define an new function $lookup_{SB}$, in Algorithm 5, which returns a set of values from matching stores in the store buffer. Additionally, $lookup_{SB}$ returns the depth at which each load must be invalidated, that is, the retirement depth of a most recent store to the same address. Notice that a redundant value is discarded whenever *lookup* is able to determine that the load and the store do not alias (case $eq^{\#}(i,j) = false$).

**Func** $lookup_{SB}(SB, Mem, i)$ **is**
  $S \leftarrow \emptyset$; ▷ *Set of load values*
  $\delta \leftarrow \infty$; ▷ *Retirement depth of preceding matching store*
  **for** $((\boldsymbol{store}(\widehat{\mu}, j, \widehat{\varphi}), \delta')$ **in** $SB)$ **do**
    **if** $eq^{\#}(i,j) = true$ **then** ▷ *Must alias*
      $S \leftarrow S \cup \{(\widehat{\varphi}, \delta)\}$;
      $\delta \leftarrow \delta'$;
    **else if** $eq^{\#}(i,j) = \bot$ **then** ▷ *May alias*
      $S \leftarrow S \cup \{(\boldsymbol{select}(\widehat{\mu}, i), \delta)\}$;
      $\delta \leftarrow \delta'$;
    **else if** $eq^{\#}(i,j) = false$ **then** ▷ *Must not alias*
      **continue**
  $S \leftarrow S \cup \{(lookup_{mem}(Mem), \delta)\}$; ▷ *Load from main memory*
  **return** $S$

**Algorithm 5:** Definition of $lookup_{SB}$

**Encode remaining cases in single expression.** Finally, we define a function $lookup_{ite}(\widehat{\mu}, i, \widetilde{\lambda}, \widetilde{\delta})$, in Algorithm 6, which encodes the result of $lookup_{SB}$ as a symbolic if-then-else expression. The function declares fresh boolean variables, and encodes all the possible values that the load can take in a single if-then-else expression. By choosing the value of the booleans variables, the solver can choose the value of the load. The function also updates the *transient load set* $\widetilde{\lambda}$, in order to invalidate transient loads when their value is overwritten by a more recent store at the same address.

**Func** $lookup_{ite}(\widetilde{\lambda}, \widehat{\mu}, i, \widetilde{\delta})$ **is**
  $S \leftarrow lookup_{SB}(SB(\widehat{\mu}, \widetilde{\delta}), Mem(\widehat{\mu}, \widetilde{\delta}), i)$; ▷ *Set of possible load values*
  $\widehat{\nu} \leftarrow get_{\infty}(S)$; ▷ *Get sequential value from S*
  $S \leftarrow S \setminus \{(\widehat{\nu}, \infty)\}$;
  **for** $(\widehat{\varphi}, \delta)$ **in** $S$ **do**
    $\beta \leftarrow fresh\_boolean\_var$;
    $\widehat{\nu} \leftarrow ite\ \beta\ then\ \widehat{\varphi}\ else\ \widehat{\nu}$;
    $\widetilde{\lambda} \leftarrow \widetilde{\lambda} \cup \{(\beta, \delta)\}$; ▷ *Save retire depth in* $\widetilde{\lambda}$
  **return** $\widehat{\nu}, \widetilde{\lambda}$

**Algorithm 6:** Definition of $lookup_{ite}$ where $get_{\infty}(S)$ returns the value in $S$ which depth matches $\infty$.

**Evaluation of load expressions.** The evaluation of load expressions is detailed in Algorithm 7 [3]. First, the function evaluates the symbolic value of the index and

3. Algorithm 7 is the translation of the rule LOAD defined in Figure 6.4 which we express here as an algorithm for simplicity

check that it can be leaked securely. Then it calls $lookup_{ite}$, which returns the set of symbolic values that the load can take, encoded as a single if-then-else expression $\hat{\iota}$ and updates the set of transient load $\widetilde{\lambda}$. Finally, it computes the retirement depth of the load (i.e. current depth + maximum speculation depth). The retirement depth is later used in the evaluation of conditional branches to determine speculation depth of conditional instructions (cf. Section 6.3.2.1).

**Func** *eval_load*$(s)$ **where** $\mathbb{P}[s.l] \ni$ `load` $i$ **is**
  $s\ i \vdash \hat{\iota}$; ▷ *Evaluates index $i$ to $\hat{\iota}$*
  **assert** $secLeak(\hat{\iota}, s.\pi)$; ▷ *Ensure that leakage of $\hat{\iota}$ is secure*
  $\hat{\nu}, \widetilde{\lambda}' \leftarrow lookup_{ite}(s.\widetilde{\lambda}, s.\hat{\mu}, \hat{\iota}, s.\widetilde{\delta})$;
  $\delta' = \widetilde{\delta} + \Delta$; ▷ *Compute retirement depth*
  **return** $(\hat{\nu}^{\delta;}, \widetilde{\lambda}')$

**Algorithm 7:** Evaluation of load expressions.

#### 6.3.3.3 Invalidate transient loads

Transient load values can be invalidated when more recent matching stores are retired by setting the corresponding boolean variables to $false$. We define a function $retire_{\text{STL}}(\pi, \widetilde{\lambda}, \widetilde{\delta})$, in Algorithm 8, that removes from the set of transient loads $\widetilde{\lambda}$ all the loads with an invalidation depth below $\widetilde{\delta}$, and set the corresponding booleans to $false$ in the path predicate $\pi$.

**Func** $retire_{\text{STL}}(\pi, \widetilde{\lambda}, \widetilde{\delta})$ **is**
  $\pi' \leftarrow \pi$;
  $\widetilde{\lambda}' \leftarrow \emptyset$;
  **for** $(\beta, \delta)$ **in** $\widetilde{\lambda}$ **do**
    **if** $\delta \leq \widetilde{\delta}$ **then**
      $\pi' \leftarrow \pi' \wedge (\beta = false)$; ▷ *Retire load*
    **else**
      $\widetilde{\lambda}' \leftarrow \widetilde{\lambda}' \cup \{(\beta, \delta)\}$
  **return** $(\pi', \widetilde{\lambda}')$

**Algorithm 8:** Retire expired load values.

### 6.3.4 Symbolic evaluation of instructions

**Stop all speculations.** For convenience, we introduce a function $retire_{\text{ALL}}$ that retires both transient paths and load values by applying both $retire_{\text{PHT}}$ and $retire_{\text{STL}}$. This function is used, for instance, at serializing instructions—like `lfence`, `mfence`, or `cpuid`—with a current depth set to $\infty$ to stop all speculations. A formal definition is given in Algorithm 9.

**Evaluation of instructions.** Symbolic evaluation of DBA instructions, denoted $s \rightsquigarrow s'$ where $s$ and $s'$ are symbolic configurations, is given in Figure 6.5. For simplicity, we omit the evaluation of indirect jumps because it relies on different speculation mechanisms than those considered in this work. The rules are given without all the optimizations for binary-level RelSE introduced in Section 4.4.2; notice however

**Func** *retire*ALL$(\pi, \widetilde{\pi}, \widetilde{\lambda}, \widetilde{\delta})$ **is**
  $(\pi', \widetilde{\pi}') \leftarrow$ *retire*PHT$(\pi, \widetilde{\pi}, \widetilde{\delta})$;
  $(\pi'', \widetilde{\lambda}') \leftarrow$ *retire*STL$(\pi', \widetilde{\lambda}, \widetilde{\delta})$;
  **return** $(\pi'', \widetilde{\pi}', \widetilde{\lambda}')$

**Algorithm 9:** Retire all expired transient values.

that these optimizations also apply to Haunted RelSE [4]. Detailed explanations of the symbolic evaluation rules follow:

— STEP-RETIRE is the rule that invalidates transient values before actually evaluating the current instruction. It retires pending conditions in $\widetilde{\pi}$ and transient loads in $\widetilde{\lambda}$ matching the current depth $\widetilde{\delta}$. It also updates the path predicate $\pi$, and checks its satisfiability, ending the path if it becomes unsatisfiable. Finally, it evaluates the current instruction (with relation $\rightsquigarrow_i$) and returns the next symbolic configuration;

— S-JUMP is the evaluation of a static jump. It moves control to the jump target and increments the current depth of the symbolic execution (like other instructions).

— FENCE is the evaluation of a serializing instruction (e.g. `lfence`, `mfence`, `cpuid`). It invalidates all transient values and transient paths by calling the function *retire*ALL with depth set to $\infty$. Finally, it checks if the path predicate is satisfiable—which terminates unsatisfiable paths;

— ITE-TRUE is the evaluation of a conditional jump when the expression is speculatively evaluated to true (the false case is analogous) [5]. First, the rule checks that the leakage of the condition is secure ($\mathit{secLeak}(\widehat{\varphi}, \pi)$). Then it adds the condition $\widehat{\varphi}$ and its retirement depth $\delta$ to the speculative path predicate $\widetilde{\pi}$ (cf. box). Finally, the rule jumps to the location indicated by the label. Notice that the condition is not added the path predicate right away. It will be added to the path predicate by the rule STEP-RETIRE when $\widetilde{\delta}$ reaches $\delta$. In particular, if $\widehat{\varphi}$ does not depend on the memory (and thus can be resolved quickly), we have $\delta \leq \widetilde{\delta}$ and the condition is retired just before evaluating the next instruction (hence there is no speculation);

— ASSIGN is the evaluation of an assignment. The rule evaluates the expression $e$ to $\widehat{\varphi}$, generates a temporary variable $\widehat{\nu}$ (in order to avoid term size explosion), and sets $\widehat{\nu}$ to $\widehat{\varphi}$ in the path predicate. Finally, it updates the variable `v` in the register map $\rho$ with its new value $\widehat{\nu}$ and its memory-dependency depth $\delta$;

— STORE is the evaluation of a store instruction [6]. The rule evaluates the index $e_{idx}$ to $\hat{\imath}$ and checks under the sequential path predicate—obtained by calling *retire*ALL with depth set to $\infty$—that leaking the index is secure ($\mathit{secLeak}(\hat{\imath}, \pi_{seq})$). This accounts for the fact that store indexes can leak in sequential execution but not in transient execution [252]. Next, it updates the path predicate $\pi$ and the relational memory $\widehat{\mu}$ with a relational store instruction. Additionally, it set the retirement depth of the store operation to $\widetilde{\delta} + \Delta$ (i.e. current depth plus size of the reorder buffer). This depth is later used in the

4. On-the-fly read over write is already included in function $\mathit{lookup}_{ite}$

5. Rule ITE-TRUE is the translation of Algorithm 2, which we formalize here as a rule for the sake of exhaustiveness.

6. Rule STORE is the translation of Algorithm 4, which we formalize here as a rule for the sake of exhaustiveness.

**Instr**

$$\text{STEP-RETIRE}\quad \frac{\boxed{\pi', \widetilde{\pi}', \widetilde{\lambda}' = \mathit{retire}_{\mathrm{ALL}}(\pi, \widetilde{\pi}, \widetilde{\lambda}, \widetilde{\delta})} \quad \vDash_{\mathrm{SMT}} \pi' \quad (l, \widetilde{\delta}, \rho, \widehat{\mu}, \pi', \widetilde{\pi}', \widetilde{\lambda}') \rightsquigarrow_i (l', \widetilde{\delta}', \rho', \widehat{\mu}', \pi'', \widetilde{\pi}'', \widetilde{\lambda}'')}{(l, \widetilde{\delta}, \rho, \widehat{\mu}, \pi, \widetilde{\pi}, \widetilde{\lambda}) \rightsquigarrow (l, \widetilde{\delta}, \rho, \widehat{\mu}, \pi'', \widetilde{\pi}'', \widetilde{\lambda}'')}$$

$$\text{S-JUMP}\quad \frac{\mathbb{P}[l] = \texttt{goto}\ l'}{(l, \widetilde{\delta}, \_) \rightsquigarrow_i (l', \widetilde{\delta} + 1, \_)}$$

$$\text{FENCE}\quad \frac{\mathbb{P}[l] = \texttt{fence} \quad \pi', \widetilde{\pi}', \widetilde{\lambda}' = \mathit{retire}_{\mathrm{ALL}}(\pi, \widetilde{\pi}, \widetilde{\lambda}, \infty) \quad \vDash_{\mathrm{SMT}} \pi'}{(l, \widetilde{\delta}, \_, \pi, \widetilde{\pi}, \widetilde{\lambda}) \rightsquigarrow_i (l, \widetilde{\delta} + 1, \_, \pi', \widetilde{\pi}', \widetilde{\lambda}')}$$

$$\text{ITE-TRUE}\quad \frac{\mathbb{P}[l] = \texttt{ite}\ e\ \texttt{?}\ l_{true}\ \texttt{:}\ l_{false} \quad (l, \widetilde{\delta}, \_, \pi, \widetilde{\pi}, \widetilde{\lambda})\ e \vdash \widehat{\varphi}^{\delta}, \widetilde{\lambda}' \quad \boxed{\widetilde{\pi}' \triangleq \{\mathit{true} = \widehat{\varphi}_{|l} = \widehat{\varphi}_{|r}, \delta\} \cup \widetilde{\pi}} \quad \mathit{secLeak}(\widehat{\varphi}, \pi)}{(l, \widetilde{\delta}, \_, \pi, \widetilde{\pi}, \widetilde{\lambda}) \rightsquigarrow_i (l_{true}, \widetilde{\delta} + 1, \_, \pi, \widetilde{\pi}', \widetilde{\lambda}')}$$

$$\text{ITE-FALSE}\quad \frac{\mathbb{P}[l] = \texttt{ite}\ e\ \texttt{?}\ l_{true}\ \texttt{:}\ l_{false} \quad (l, \widetilde{\delta}, \_, \pi, \widetilde{\pi}, \widetilde{\lambda})\ e \vdash \widehat{\varphi}^{\delta}, \widetilde{\lambda}' \quad \boxed{\widetilde{\pi}' \triangleq \{\mathit{false} = \widehat{\varphi}_{|l} = \widehat{\varphi}_{|r}, \delta\} \cup \widetilde{\pi}} \quad \mathit{secLeak}(\widehat{\varphi}, \pi)}{(l, \widetilde{\delta}, \_, \pi, \widetilde{\pi}, \widetilde{\lambda}) \rightsquigarrow_i (l_{\mathit{false}}, \widetilde{\delta} + 1, \_, \pi, \widetilde{\pi}', \widetilde{\lambda}')}$$

$$\text{ASSIGN}\quad \frac{\mathbb{P}[l] = \mathtt{v}\ \texttt{:=}\ e \quad (l, \widetilde{\delta}, \rho, \_, \pi, \_, \widetilde{\lambda})\ e \vdash \widehat{\varphi}^{\delta}, \widetilde{\lambda}' \quad \widehat{\varphi}' = \mathit{fresh}(\widehat{\varphi}) \quad \rho' \triangleq \rho[\mathtt{v} \mapsto \widehat{\varphi}'^{\delta}] \quad \pi' \triangleq \pi \wedge (\widehat{\varphi}'_{|l} = \widehat{\varphi}_{|l}) \wedge (\widehat{\varphi}'_{|r} = \widehat{\varphi}_{|r})}{(l, \widetilde{\delta}, \rho, \_, \pi, \_, \widetilde{\lambda}) \rightsquigarrow_i (l + 1, \widetilde{\delta} + 1, \rho', \_, \pi', \_, \widetilde{\lambda}')}$$

$$\text{STORE}\quad \frac{\begin{array}{c}\mathbb{P}[l] = \texttt{store}\ e_{idx}\ e_{val} \\ (l, \widetilde{\delta}, \_, \widehat{\mu}, \pi, \widetilde{\pi}, \widetilde{\lambda})\ e_{idx} \vdash \widehat{\imath}, \widetilde{\lambda}' \quad (l, \widetilde{\delta}, \_, \widehat{\mu}, \pi, \widetilde{\pi}, \widetilde{\lambda}')\ e_{val} \vdash \widehat{\nu}, \widetilde{\lambda}'' \\ \widehat{\mu}' \triangleq (\boldsymbol{store}(\widehat{\mu}, \widehat{\imath}, \widehat{\nu}), \boxed{\widetilde{\delta} + \Delta}) \quad \pi' \triangleq (\pi \wedge \widehat{\mu}' = \boldsymbol{store}(\widehat{\mu}, \widehat{\imath}, \widehat{\nu})) \\ (\pi_{seq}, \_, \_) \triangleq \mathit{retire}_{\mathrm{ALL}}(\pi, \widetilde{\pi}, \widetilde{\lambda}, \infty) \quad \mathit{secLeak}(\widehat{\imath}, \boxed{\pi_{seq}})\end{array}}{(l, \widetilde{\delta}, \_, \widehat{\mu}, \pi, \widetilde{\pi}, \widetilde{\lambda}) \rightsquigarrow_i (l + 1, \widetilde{\delta} + 1, \_, \widehat{\mu}', \pi', \widetilde{\pi}, \widetilde{\lambda}'')}$$

FIGURE 6.5 – Relational symbolic evaluation of DBA instructions and expressions using Haunted RelSE, where $\mathit{fresh}(\widehat{\varphi})$ returns a pair of fresh symbolic variables if $\widehat{\varphi}$ is a pair, or a simple fresh symbolic variable if $\widehat{\varphi}$ is simple and $\boldsymbol{store}$ is the lifting of a symbolic store operation to relational expressions. For readability, terms that are not important in the context are replaced with _ and and important details are highlighted with $\boxed{\mathit{boxes}}$.

function $\mathit{lookup}_{SB}$ to determine which store operations are retired and which store operations might still be in the store buffer.

### 6.3.5 Theorems

In this section we prove that Haunted RelSE is correct and complete (up-to-an-unrolling-bound) for SCT. This means that when Haunted RelSE reports a violation, it is a real violation of SCT (no over-approximation); and when it reports no violations up to depth $k$ then the program is secure up to depth $k$ (no under-approximation). To this end, we prove that Haunted RelSE is equivalent to Explicit RelSE (Theorem 6) and show that Explicit RelSE is correct and complete up-to-an-unrolling-bound for SCT (Theorem 5).

**Theorem 5** (Correct and complete Explicit RelSE)**.** *Explicit RelSE is correct and complete up-to-an-unrolling-bound for speculative constant-time.*

PROOF (SKETCH). The proof is a simple extension of the proofs of correct bug-finding and bounded-verification of RelSE for constant-time (Theorem 2 and Theorem 4), to the speculative semantics. The extension requires to show that:

1. Violations reported on transient paths in the symbolic execution correspond to violations in concrete transient execution (correct bug-finding);
2. If there is a violation in concrete transient execution, then there is a path in symbolic execution that reports this violation (correct bounded-verification).

□

Next, we show that Haunted RelSE is equivalent to Explicit RelSE.

**Theorem 6** (Equivalence Explicit and Haunted RelSE)**.** *Haunted RelSE detects a violation in a program if and only if Explicit RelSE detects a violation.*

A sketch a proof is given in Appendix A.2. We first show that the theorem holds for Spectre-PHT: after a conditional branch, the two paths explored in Haunted RelSE exactly capture the behavior of the four paths explored in Explicit RelSE. Then, we show that it holds for Spectre-STL: after a load instruction, the single path resulting from Haunted RelSE exactly captures the behavior of the multiple paths explored in Explicit RelSE.

**Corollary 1.** *Haunted RelSE is correct and complete up-to-an-unrolling-bound for speculative constant-time.*

## 6.4 Implementation

We implement Haunted RelSE on top of the binary-level analyzer BINSEC/REL (Section 4.5) in a tool named BINSEC/HAUNTED[7]. BINSEC/HAUNTED takes as input an `x86` executable, the location of secret inputs, an initial memory configuration (possibly fully symbolic), the maximum speculation depth $\Delta$, and the size of the store buffer $|SB|$. BINSEC/HAUNTED explores the program in a depth-first search manner, prioritizing transient paths over sequential paths, and uses the SMT solver Boolector [191], currently the best for the theory of bitvectors [233, 107].

BINSEC/HAUNTED reports SCT violations, together with a counterexamples (i.e., initial configurations and speculation choices leading to the violation). Notably, BINSEC/HAUNTED uses the name of the boolean variables encoding load values as *ite*-expressions, to encode information about the location of the load and the forwarding store (more details in Appendix B.3). Therefore, using the counterexample returned

7. Open sourced at: https://github.com/binsec/haunted

by the solver it is possible to understand which stores have been bypassed to trigger the violation.

Finally, the validation of BINSEC/HAUNTED for detecting Spectre vulnerabilities is challenging because there is no ground truth (especially for Specter-STL) and it is difficult to manually reason about SCT violations. For Spectre-PHT, we validate BINSEC/HAUNTED against an existing set of insecure litmus test [154], and a version that we patch using index-masking [113]. For Spectre-STL, we manually crafted and documented a new set of 14 litmus tests [8] (an excerpt is given in Appendix C.2.3). Our results are cross-checked against two other tools and manually checked in case of deviation. More details on the validation of BINSEC/HAUNTED are given in Appendix B.4.

## 6.5 Experimental evaluation

**Section overview**

This section presents the experimental evaluation of our technique *Haunted RelSE* and tool BINSEC/HAUNTED. It first details the research questions we address and our methodology (cf. Section 6.5.1). Next, it evaluates performance of Haunted RelSE against the standard approach, Explicit RelSE (also implemented as a part of BINSEC/HAUNTED) for Spectre-PHT (cf. Section 6.5.2), and for Spectre-STL (cf. Section 6.5.2). Finally, it compares BINSEC/HAUNTED against two state-of-the-art tools (cf. Section 6.5.4).

### 6.5.1 Research questions and methodology

**Research questions.** To assess the performance of our technique, *Haunted RelSE*, and tool, BINSEC/HAUNTED, we outline the following research questions:

**RQ1 Effectiveness.** Is BINSEC/HAUNTED able to find Spectre-PHT and Spectre-STL violations in real-world cryptographic binaries?

**RQ2 Haunted vs. Explicit.** How does *Haunted RelSE* compares against *Explicit RelSE*?

**RQ3 BINSEC/HAUNTED vs. SoA tools.** How does BINSEC/HAUNTED compare against state-of-the-art tools?

To answer **RQ1** and **RQ2**, we compare the performance of *Explicit* and *Haunted* explorations strategies for RelSE—both implemented in BINSEC/HAUNTED—on a set of real word cryptographic binaries and litmus benchmark (for Spectre-PHT in Section 6.5.2, and for Spectre-STL in Section 6.5.3). To answer **RQ3**, we compare BINSEC/HAUNTED against state-of-the-art competitors, KLEESpectre [252] and Pitchfork [67] (Section 6.5.4).

**Legend.** We evaluate performance in terms of:

— Number of *unique* x86 instructions explored ($I_{x86}$)—gives an indication of the coverage of the analysis,

— Number of paths explored (Paths)—gives an indication on path explosion,

— Overall execution time (Time),

8. Open sourced at https://github.com/binsec/haunted_bench/blob/master/src/litmus-stl/programs/spectrev4.c

— Number of violations (🐞), i.e. the number instructions leaking secret data,

— Number of timeouts (⌛),

— Number of programs proven secure (✓),

— Number of programs proven insecure (✗).

These metrics give a good overview of the efficiency ($I_{x86}$, Paths, Time, ⌛) and effectiveness (🐞, ✓, ✗) of the analysis.

**Setup.** Experiments were performed on a laptop with an Intel(R) Xeon(R) CPU E3-1505M v6 @ 3.00GHz processor and 32GB of RAM. In the experiments, all inputs are symbolic except for the initial stack pointer `esp` (similar as related work [67]), and data structures are statically allocated. The user is expected to label secrets, all other values are public. We set the speculation depth $\Delta$ to 200 instructions and the size of the store buffer $|SB|$ to 20 instructions, which correspond to realistic values in modern processors.

Additionally, we only consider indirect jump targets resulting from *in-order* execution and implement a *shadow stack* to constrain return instructions to their proper return site. Considering transient jump targets requires to model indirect jumps on arbitrary locations, which is doable but intractable for symbolic execution.

**Benchmark.** We evaluate BINSEC/HAUNTED on the following programs:

— `litmus-pht`: 16 small test cases (litmus tests) for Spectre-PHT taken from Pitchfork, which are modified versions of Paul Kocher's litmus tests [95] to be constant-time in sequential execution,

— `litmus-pht-patched`: `litmus-pht` that we patched with index masking [113],

— `litmus-stl`: our new set of litmus tests for Spectre-STL, [9],

— Cryptographic primitives from OpenSSL and Libsodium cryptographic libraries (detailed in Table 6.2), including and extending those analyzed in [67].

Programs are compiled statically for a 32-bit x86 architecture with `gcc 10.1.0`. Litmus tests are compiled with options `-fno-stack-protector` and Spectre-STL litmus tests are additionally compiled with `-no-pie` and `-fno-pic` in order to rule out violations introduced by these options (see Section 6.6). For the same reason, `donna` and `tea` are compiled without stack protectors `-fno-stack-protector` and for optimization levels `O0`, `O1`, `O2`, `O3`, and `Ofast`. Libsodium is compiled with the default Makefile and OpenSSL is compiled with optimization level `O3` (both including stack protector).

*Note on Stack Protectors:* Error-handling code introduced by stack protectors is complex and contains many syscalls that cannot be analyzed directly in pure symbolic execution. BINSEC/HAUNTED stops path execution on syscalls and only jump on the error-handling code of stack protectors once per program, meaning that it might miss violations in unexplored parts of the code. Moreover, timeout is set to 1 hour for litmus tests, `tea`, and `donna`; but extended to 6 hours for code containing stack protectors (Libsodium and OpenSSL).

### 6.5.2 Performance for Spectre-PHT (RQ1-RQ2)

We compare the performance of Haunted RelSE and Explicit RelSE—that we call Haunted and Explicit in the tables for brevity—for detecting Spectre-PHT violations.

9. Open sourced at: https://github.com/binsec/haunted_bench

| Programs | Type | $I_{x86}$ | Key | Msg |
|---|---|---|---|---|
| `tea_encrypt` [255] | Block cipher | 100 | 16 | 8 |
| `curve25519-donna` [167] | Elliptic curve | 5k | 32 | - |
| Libsodium `secretbox` [226] | Stream cipher | 3k | 32 | 256 |
| OpenSSL `ssl3-digest-rec` [9] | HMAC | 2k | 32 | 256 |
| OpenSSL `mee-cbc-decrypt` [9] | MEE-CBC | 6k | 16+32 | 64 |

TABLE 6.2 – Cryptographic benchmarks, with approximate static instruction count ($I_{x86}$) (*excluding libc code*) and sizes of secret keys and messages (Msg) in bytes.

In order to focus on Spectre-PHT only, we disable support for Spectre-STL. Additionally, we also report the performance for standard constant-time verification (without speculation) as a baseline, called NoSpec. Results are presented in Table 6.3. To show the importance of Haunted RelSE for path pruning in programs containing loops, we also detail the execution of a litmus test containing a loop (`case_5`) in Appendix C.2.2.

**Results.** For `litmus-pht` and `litmus-pht-masked`, we can see that Haunted RelSE:

— explores fewer paths (4×) for an equivalent result, limiting path explosion (see Appendix C.2.2),

— analyzes programs faster (1437× and 21× respectively), achieving performance in line with NoSpec,

— can fully explore 2 additional programs and finds 1 more violation whereas Explicit RelSE times-out.

For `tea` and `donna` there is no difference between Explicit and Haunted. Indeed, because these programs only have a single feasible path in sequential execution, Explicit RelSE forks into two paths at each conditional branch instead of four (the two other paths being unsatisfiable) which makes it equivalent to Haunted RelSE.

Finally, for Libsodium and OpenSSL, Explicit RelSE gets stuck exploring complex code introduced by stack protectors and spends most of its time checking satisfiability of the path predicate before timing out. Haunted RelSE circumvents this issue by delaying the update of the path predicate, thus it can fully explore `secretbox` and `ssl3-digest` without timing-out, with a noticeable speedup (8.9× and 4.6×), covering more code (4.6× and 3×), and finding 4 more violations. While Haunted RelSE times out on the more complex primitive `mee-cbc`, it still explores 3.5× more code than Explicit.

**Conclusion.** While the Explicit strategy already allows to find Spectre-PHT violations in realistic codes, Haunted RelSE strongly improves the performance in terms of speed (2.3× faster in total and up to 1437× on `litmus-pht`), timeouts (-66%) and covered code (1.28× in total and up to 4.6× for `secretbox`). We can see that Haunted RelSE does not improve performance over Explicit RelSE in 3/7 use-cases, but make a noticeable difference on the other 4/7 use-cases (`litmus-pht`, `litmus-pht masked`, `secretbox`, `ssl3-digest`), where the performance gains become significant (from 4.6× faster to 1437×).

| Programs | PHT | $I_{x86}$ | Paths | Time | 🐞 | ⌛ | ✓ | ✗ |
|---|---|---|---|---|---|---|---|---|
| `litmus-pht` | NoSpec | 733 | 48 | 3 | - | 0 | 16/16 | - |
| | Explicit | 761 | **703** | **10331** | **21** | **2** | - | 16/16 |
| | Haunted | 761 | **188** | **7** | **22** | **0** | - | 16/16 |
| `litmus-pht masked` | NoSpec | 915 | 48 | 5 | - | 0 | 16/16 | - |
| | Explicit | 950 | **843** | **169** | - | 0 | 16/16 | - |
| | Haunted | 950 | **182** | **8** | - | 0 | 16/16 | - |
| `tea` | NoSpec | 326 | 5 | .56 | - | 0 | 5/5 | - |
| | Explicit | 326 | 172 | .62 | - | 0 | 5/5 | - |
| | Haunted | 326 | 172 | .62 | - | 0 | 5/5 | - |
| `donna` | NoSpec | 22k | 5 | 2948 | - | 0 | 5/5 | - |
| | Explicit | 21k | 1.0M | 6153 | - | 1 | 4/5 | - |
| | Haunted | 21k | 1.0M | 6162 | - | 1 | 4/5 | - |
| `secretbox` | NoSpec | 2721 | 1 | 5 | - | 0 | 1/1 | - |
| | Explicit | **769** | 15k | **21600** | **13** | **1** | - | 1/1 |
| | Haunted | **3583** | 2.2M | **2421** | **17** | **0** | - | 1/1 |
| `ssl3-digest` | NoSpec | 1809 | 1 | 4 | - | 0 | 1/1 | - |
| | Explicit | **808** | 9k | **21600** | 13 | **1** | - | 1/1 |
| | Haunted | **2502** | 428k | **4694** | 13 | **0** | - | 1/1 |
| `mee-cbc` | NoSpec | 6383 | 1 | 448 | - | 0 | 1/1 | - |
| | Explicit | **696** | 74k | 21600 | 17 | 1 | - | 1/1 |
| | Haunted | **2549** | 22M | 21600 | 17 | 1 | - | 1/1 |
| Total | NoPHT | 35k | 109 | 3415 | 0 | 0 | 45/45 | - |
| | Explicit | **25k** | 1.1M | **81453** | **64** | **6** | 25/25 | 19/19 |
| | Haunted | **32k** | 25.7M | **34892** | **69** | **2** | 25/25 | 19/19 |

TABLE 6.3 – Performance of BINSEC/HAUNTED for Spectre-PHT.

### 6.5.3 Performance for Spectre-STL (RQ1-RQ2)

We compare the performance of Haunted RelSE and Explicit RelSE for detecting Spectre-STL violations. In order to focus on Spectre-STL only, we disable support for Spectre-PHT. Results are presented in Table 6.4.

**Results.** The explosion of the number of paths for Explicit RelSE shows that the number of behaviors to consider for Spectre-STL grows exponentially. The performance of Explicit RelSE on litmus tests shows that encoding transient paths explicitly is not tractable—even though our implementation discards redundant paths. Overall, Haunted RelSE scales better on `litmus-stl` tests and `tea`, achieving better analysis time (speed up of 3152× and 3.4×), producing fewer timeouts (0 vs. 7), and finding more violations (+24). While it times out on more complex code, it explores much more instruction than Explicit RelSE (8.6× more unique instructions in total), finds 126 more violations and reports 10 more insecure programs.

**Conclusion.** While the state-of-the-art Explicit strategy shows low performance for Spectre-STL even on small programs, Haunted RelSE strongly improves the performance in terms of:

| Programs | STL | $I_{x86}$ | Paths | Time | 🐞 | ⌛ | ✓ | ✗ |
|---|---|---|---|---|---|---|---|---|
| litmus-stl | NoSpec | 328 | 14 | .5 | - | 0 | 14/14 | - |
| | Explicit | 316 | **37M** | **7205** | **13** | **2** | **3/4** | 10/10 |
| | Haunted | 328 | **14** | **2.3** | **13** | **0** | **4/4** | 10/10 |
| tea | NoSpec | 326 | 5 | .5 | - | 0 | 5/5 | - |
| | Explicit | 278 | **12M** | **18000** | **2** | **5** | - | **1/5** |
| | Haunted | 326 | **18** | **5276** | **26** | **0** | - | **5/5** |
| donna | NoSpec | 22k | 5 | 2948 | - | 0 | 5/5 | - |
| | Explicit | **704** | 12M | 18000 | **0** | 5 | - | **0/5** |
| | Haunted | **12k** | 5 | 18000 | **73** | 5 | - | **5/5** |
| secretbox | NoSpec | 2721 | 1 | 5 | - | 0 | 1 | - |
| | Explicit | **225** | 13M | 21600 | **4** | 1 | - | **1/1** |
| | Haunted | **408** | 2 | 21600 | **26** | 1 | - | **1/1** |
| ssl3-digest | NoSpec | 1809 | 1 | 4 | - | 0 | 1/1 | - |
| | Explicit | **204** | 4k | 21600 | **3** | 1 | - | 1/1 |
| | Haunted | **1763** | 2 | 21600 | **8** | 1 | - | 1/1 |
| mee-cbc | NoSpec | 6383 | 1 | 448 | - | 0 | 1/1 | - |
| | Explicit | **200** | 19M | 21600 | **0** | 1 | - | **0/1** |
| | Haunted | **1627** | 1 | 21600 | **2** | 1 | - | **1/1** |
| Total | NoSpec | 34k | 27 | 3407 | - | 0 | 27 | - |
| | Explicit | **2k** | 93M | 108004 | **22** | **15** | **3/4** | **13/23** |
| | Haunted | **17k** | 42 | 88078 | **148** | **8** | **4/4** | **23/23** |

TABLE 6.4 – Performance of BINSEC/HAUNTED Spectre-PHT.

— speed (1.2× faster in total and up to 3152× on `litmus-stl`),

— timeouts (8 vs. 15),

— covered code (8.6× more instructions covered in total),

— number of violation found (+126),

— and number of programs deemed insecure (+10).

Especially, Haunted RelSE manages to fully explore small-size real-world cryptographic implementations (up to one hundred instructions) and to find violations in medium-size real-world cryptographic implementations (a few thousands instructions).

### 6.5.4 Comparison with Pitchfork and KLEESpectre (RQ3)

We compare BINSEC/HAUNTED against two state-of-the-art competitors, KLEESpectre [252] and Pitchfork [67]. We discuss in more details the challenges of this comparison and the solutions we adopted (when applicable) in Appendix C.1.

**KLEESpectre.** KLEESpectre [252] is an adaptation of SE for finding Spectre-PHT violations [10], following an *Explicit exploration strategy*. It is based on the popular dynamic symbolic execution platform KLEE [58]. While Pitchfork and BINSEC/HAUNTED analyze binary code, KLEESpectre analyses LLVM bitecode, which gives it a performance advantage. Note that KLEESpectre reports several types of

10. It also includes cache modeling—disabled for our comparison.

gadgets but only one—leak secret (LS)—can actually leak secret data and is a violation of speculative constant-time, thus we only report LS gadgets found by KLEESpectre. Additionally, it does not report leakage from insecure branches.

**Pitchfork.** Pitchfork [67] is the only competing tool which can analyze programs for Spectre-STL. It is build on top of the popular binary analysis platform `angr` [230]. It is based on SE and *tainting* which is *faster than RelSE but also less precise* and can report false alarms (see Section 6.7). Pitchfork stops a path after finding a violation, whereas BINSEC/HAUNTED continues the execution. To provide a fair comparison, we also consider a modified version of Pitchfork, namely Pitchfork-cont, which does not stop after finding a violation.

| | Programs | Tool | Time | ⧗ | 🐞 | ✓ | ✗ |
|---|---|---|---|---|---|---|---|
| PHT | `litmus-pht` | KLEESpectre | **1817** | 0 | 16 | 2† | 14/16 |
| | | Pitchfork | 1.7 | 0 | 17 | - | 16/16 |
| | | Pitchfork-cont | 6.2 | 0 | 22 | - | 16/16 |
| | | BINSEC/HAUNTED | **7.2** | 0 | 22 | - | 16/16 |
| | `litmus-pht masked` | KLEESpectre | **1751** | 0 | 0 | 16/16 | - |
| | | Pitchfork | 10.2 | 0 | 0 | 16/16 | - |
| | | Pitchfork-cont | 10.2 | 0 | 0 | 16/16 | - |
| | | BINSEC/HAUNTED | **7.8** | 0 | 0 | 16/16 | - |
| | `tea` | KLEESpectre | .4 | 0 | 0 | 5/5 | - |
| | | Pitchfork | **29.5** | 0 | 0 | 5/5 | - |
| | | Pitchfork-cont | **29.7** | 0 | 0 | 5/5 | - |
| | | BINSEC/HAUNTED | **.6** | 0 | 0 | 5/5 | - |
| | `donna` | KLEESpectre | 7825 | 1 | 0 | 4/5 | - |
| | | Pitchfork | TO | 5 | 0 | 0/5 | - |
| | | Pitchfork-cont | TO | 5 | 0 | 0/5 | - |
| | | BINSEC/HAUNTED | **6162** | 1 | 0 | 4/5 | - |
| STL | `litmus-stl` | Pitchfork | **21608*** | **6** | **11** | **1/4** | **9/10** |
| | | Pitchfork-cont | **21610*** | **6** | **11‡** | **1/4** | **9/10** |
| | | BINSEC/HAUNTED | **2.3** | **0** | **13** | **4/4** | **10/10** |
| | `tea` | Pitchfork | TO | **5** | **0** | - | **0/5** |
| | | Pitchfork-cont | TO | **5** | **0** | - | **0/5** |
| | | BINSEC/HAUNTED | **5275** | **0** | **26** | - | **5/5** |
| | `donna` | Pitchfork | TO | 5 | 0 | - | **0/5** |
| | | Pitchfork-cont | TO | 5 | 0 | - | **0/5** |
| | | BINSEC/HAUNTED | TO | 5 | **73** | - | **5/5** |

TABLE 6.5 – Performance of BINSEC/HAUNTED, Pitchfork and KLEESpectre on `tea`, and Spectre-PHT and Spectre-STL litmus tests. Timeout (⧗) is set to 1 hour. †False positives. ‡Excluding 6 spurious violations in (non executable) `.data` section. *Excluding ⧗, times are respectively 8.1 and 10.6.

**Setup.** Performance of KLEESpectre, Pitchfork, Pitchfork-cont and BINSEC/HAUNTED on `litmus-pht`, `litmus-pht-masked`, `tea`, and `donna` are reported

in Table 6.5. We exclude `secretbox`, `ssl3-digest` and `mee-cbc` as the performance of the tools on these programs will vary according to how they handle syscalls. [11] We report unique violations for each tool. We also exclude 6 spurious violations found by Pitchfork in non executable `.data` section after following a transient indirect jump.

For Spectre-PHT, we set speculation window to 200 in all tools—which corresponds to a realistic speculation window in modern processors. For Specte-STL, we set the size of the store buffer to 20 in BINSEC/HAUNTED (thus a load can bypass up to 20 stores in a window of 200 instruction); whereas Pitchfork has less realistic speculation window and *only supports loads and store reordering in a window of 20 instructions.*

**Results.** KLEESpectre, as expected, shows similar results as Explicit RelSE in Table 6.3: it is slightly faster than BINSEC/HAUNTED on `tea` (1.5×), but slower on `litmus-pht` (250×) and on `litmus-pht-masked` (224×). Also, it *fails to report 2 insecure litmus tests*: `case_7` and `case_10`. Program `case_10` contains an insecure branch but KLEESpectre does not report leakage from insecure branches. Still, `case_7` contains a leak secret (LS) violation that KLEESpectre should report.

For Spectre-PHT, Pitchforks does not seem to follow an Explicit exploration strategy as it scales well on litmus tests. Pitchfork-cont is slightly faster than BINSEC/HAUNTED (1.2×) on `litmus-pht`, but it is 50× slower on `tea` and times-out on `donna`.

For Spectre-STL however, Pitchfork follows the explicit strategy which quickly leads to state explosion, poorer performance and more timeouts. The analysis even runs out-of-memory—taking 32GB of RAM—for six cases of `litmus-stl`, 1 `tea`, and 4 `donna`. Hence, Pitchfork does not scale for Spectre-STL even on small-size binaries whereas our tool can exhaustively explore small-size binaries. Our results further show that BINSEC/HAUNTED finds 112 more Spectre-STL violations, identifies 11 more insecure programs and establishes security of 3 more programs compared to Pitchfork.

## 6.6 New vulnerabilities and mitigations

**Section overview**

This section, reports on two new vulnerabilities:

— Potential problems with index-masking, a well-known defense against Spectre-PHT, and proposes correct implementations to avoid them (cf. Section 6.6.1);

— Potential vulnerabilities introduced by a popular `gcc` options to generate position-independent code (cf. Section 6.6.2).

This section also confirms vulnerabilities with stack protectors and function returns, already reported by Cauligi et al. [67] (cf. Section 6.6.3).

Programs are compiled with `gcc-10.2.0 -m32 -march=i386 -O0`. All vulnerabilities were automatically found by BINSEC/HAUNTED.

11. In particular, KLEESpectre and Pitchfork can respectively rely on KLEE and `angr` system call handlers whereas BINSEC does not implement system call handlers, making the comparison impossible on these programs. However, even if syscall handlers were available, implementation choices such as concretization vs. abstractions could impact the result of the analysis making the comparison challenging.

### 6.6.1 Index-masking defense

**Index-masking.** Index-masking [113] is a well known defense against Spectre-PHT—used in WebKit for example—which consists in strengthening conditional array bound checks with branchless bound checks. Indexes are masked with the length of the array, rounded up to the next power of two minus one. We give an example of index masking in Listing 6.1. For the array `publicarray` of size 16 the value of the mask is 15 (`0x0f`). For an arbitrary index `idx`, the masked index `(idx & 0x0f)` is strictly smaller than 16, hence the access is in bounds. This countermeasure prevents out-of-bound reads if the length of the array is a power of two and limits the scope of out-of-bound reads otherwise.

```
1 void leakThis(uint8_t toLeak) {
2   tmp &= publicarray2[toLeak * 512];
3 }
4 void case_1_masked(uint32_t idx) {
5   idx = idx & (publicarray_size - 1);
6   uint8_t toLeak = publicarray[idx];
7   leakThis(toLeak);
8 }
```

LISTING 6.1 – Illustration of index-masking

**Spectre-STL vulnerability.** Using BINSEC/HAUNTED, we discover that whereas this countermeasure does protect against Spectre-PHT, it may also introduce new Spectre-STL vulnerabilities. Take for instance the compiled version of Listing 6.1, given in Listing 6.2. Line 1 computes the value of the mask and store it into `eax`. Line 2 performs the index masking and stores the masked index in the memory at `[ebp + idx]`. Line 3 loads the masked index into `eax`. *Notice that this load can bypass the store at line 2 and load the old unmasked index* `idx`. Then, line 3 loads the value at `publicarray[idx]` into `al`, allowing the attacker to read arbitrary memory—including secret data. Finally, the value of `al` is used as a load index at Line 4, encoding secret data in the cache. To conclude, because the masked index is stored in the memory, the masking operation can be bypassed with Spectre-STL, leading to arbitrary memory read, and eventually leaking secret data.

```
1 mov eax, publicarray_size - 1   ; Compute mask
2 and [ebp + idx], eax            ; Store masked index
3 mov eax, [ebp + idx]            ; Bypass prior store
4 mov  al, [@publicarray + eax]   ; Out-of-bound load
5 mov  dl, publicarray2[al << 9]  ; Leak secret
```

LISTING 6.2 – Compiled version of Listing 6.1 with `gcc-10.2.0 -m32 -march=i386 -O0`

**Mitigation.** This violation of SCT occurs at optimization level `O0` with both `clang-11.0` and `gcc-10.2` because the masked index is stored on the stack. We propose a *patched implementation* in Listing 6.3 that forces the index into a register (line 2) so the masking cannot be bypassed. A second solution is to set the optimization level to `O1` or higher so the store operation is optimized away—but this solution

is fragile as it still relies on compiler choices. In these two case, BINSEC/HAUNTED reports that the program is secure with regard to speculative constant-time.

```
1 void case_1_masked_patched(uint32_t idx) {
2   register uint32_t ridx asm ("edx");
3   ridx = idx & (publicarray_size - 1);
4   uint8_t toLeak = publicarray[ridx];
5   leakThis(toLeak);
6 }
```

LISTING 6.3 – Patch of index-masking for Spectre-STL

### 6.6.2 Position-independent code

**Position-independent code.** Position-independent code (PIC), and position-independent executables (PIE) are compiler options which makes it possible to load a binary to any memory location without modifying the code. These options are used to enable address space layout randomization (ASLR), which loads executables to non-predictable addresses in order to prevent a attackers from guessing target addresses, making return oriented programming (ROP) attacks more challenging. Our version of `gcc-10` compiles by default to position independent executables, which can be disabled by adding the options `-fno-pic -no-pie`.

**Spectre-STL vulnerability.** Using BINSEC/HAUNTED, we have discovered that the code introduced by `gcc` in position independent executables *may introduce Spectre-STL vulnerabilities*. Indeed, on our set of STL-litmus-tests compiled with `-no-pie -fno-pic`, BINSEC/HAUNTED finds 13 violations and reports 4 programs as secure and 10 as insecure; whereas on STL-litmus-tests compiled without these options, it *finds 26 violations and reports only one program as secure*.

In x86, position independent executables access global variables as an offset from a *global pointer* which is set up at the beginning of the function, relatively to the current location. The current location is not directly accessible but is obtained via a function `x86_get_pc_thunk_ax` which loads its return address to `eax`. More precisely, a call to `x86_get_pc_thunk_ax` stores the return address on the stack before jumping to the function, then in the function this return address is loaded into `eax`. *With Spectre-STL, this load can bypass the previous store and load a stale value into* `eax`. Because `eax` is later used as a global offset, controlling its value, gives an attacker the ability to speculatively read at an arbitrary address. Take as an example the program in Listing 6.4, that we explain line per line:

Line 6: Call the function `x86_get_pc_thunk_ax`, and store return address to the stack;

Line 2: Load `[esp]` bypasses the previous store and gets its value from main memory; which can be populated with attacker controlled values. Here, let `eax` take the transient value `0x023f35`;

Line 7: Computes the *global pointer* for PIC using the transient value in `eax`;

Line 8: The value in `eax`—controlled by the attacker—is used as an offset to access the global variable `publicarray_size`. Consequently, secret data at address `0xC20EF` is loaded to `edx`;

Line 11: Finally, the value of the secret in `edx` is used as index for a load, which violates speculative constant-time.

```
1  __x86_get_pc_thunk_ax:
2    mov eax, [esp+0] ; bypass stored @ret and load attacker
3    retn             ; controlled value 0x023f35
4
5  case_1_masked_patched:
6    call __x86_get_pc_thunk_ax ; eax = 0x023f35
7    add  eax, 0x9E0FA          ; eax = 0x0c202f
8    mov  edx, (publicarray_size - 0x0A2000)[eax]
9    ; edx = [0x0C20EF] = secret
10   [...]
11   mov  dl, (publicarray - 0x0A2000h)[eax + edx]
12   ; Violation: secret dependent load
```

LISTING 6.4 – Compiled version of Listing 6.3, with PIC enabled. Secret data is stored at address `0xC20EF` and `publicarray_size` at address `0x0A20C0`.

### 6.6.3 Stack protectors and stale returns

We confirm two vulnerabilities with stack protectors and function returns that that have already been reported by Cauligi et al. [67]. We only discuss them superficially, interested reader can refer to [67] for more details.

**Stack protectors.** Similarly as what Cauligi et al. [67] reported, we do not directly find violations of Spectre-PHT in cryptographic primitives. However, the code for *stack protectors*, introduced by compiler to check for buffer overflows does introduce vulnerabilities. Stack protectors add a guard at the beginning of vulnerable functions which is checked when the function exits—with a conditional branch. If this conditional branch is mispredicted, the program execute the error tampering code which contains additional conditional branches that can be mispredicted, and eventually leaks secret data.

**Function returns.** When a function returns, it loads its return address from the stack before jumping on it. With Spectre-STL, it is possible for a `ret` instruction to bypass the store instruction that pushed the return address on the stack and load a stale return address. This vulnerability enables arbitrary speculative code execution and ROP-like attacks to Spectre gadgets [153].

## 6.7 Related work

**Section overview**

Related work on Spectre attacks has been discussed in Section 3.3. This section, further discusses the closest related work. We refer the interested reader to an excellent survey by Canella et al. [65] for a more general discussion on transient execution attacks and defenses.

**Speculative constant-time.** Constant-time programming is often used in cryptographic code in order to prevent side-channel timing attacks [35]. Since the advent of

microarchitectural attacks in 2018, a few works have extended this property to speculations [71, 127, 129]. We use in our work the property of speculative constant-time from Cauligi et al. [67].

**Relational symbolic execution.** Relational symbolic execution [105] offers a more precise analysis than other techniques such as tainting. For instance, Pitchfork [67], which is based on tainting, reports a violation in Listing 6.5, line 9 because `toLeak` is tainted with secret data, whereas the program is secure because `toLeak` is set to 0 before being leaked. In contrast, BINSEC/HAUNTED, based on relational symbolic execution, does not report such false alarms.

```
1  void leakThis(uint8_t toLeak) {
2    tmp &= publicarray2[toLeak * 512];
3  }
4  void case_1(uint32_t idx) {
5    if (idx < publicarray_size) {
6      uint8_t toLeak = publicarray[idx];
7      toLeak = toLeak & 0xf0;
8      toLeak = toLeak & 0x0f; // toLeak = 0
9      leakThis(toLeak);       // Leaks value 0
10 }}
```

LISTING 6.5 – Program secure to Spectre-PHT

Four previous works have used symbolic execution for analysis of cache side-channels [254, 238, 49, 87]—including our work, presented in Chapter 4. Three of them [254, 238, 87] target binary code; only two of them [49, 87] scale to real cryptographic binaries; and none of them is able to detect Spectre attacks.

**Analyses for Spectre detection.** Several tools have been proposed in the literature to detect Spectre vulnerabilities [261, 132, 252, 253, 71, 129, 67] both at LLVM level and binary level. See Table 6.6 for a comparison. On one hand, analyzers at LLVM level scale well as to analyze real cryptographic code. Unfortunately, as shown in our experiments (Sections 4.6.2.3 and 6.6) and prior works [67, 231], compilers too often introduce constant-time violations. On the other hand, tools at binary level are more challenging to develop and are often ineffective on real code due to scalability issues.

Analysis tools for Spectre are based on *static analysis* using abstract interpretation [261], model checking [71], symbolic execution [129, 67, 252, 132] and tainting [253, 67]. KLEESpectre [252] and SpecuSym [132] are built on top of KLEE [58] and Pitchfork [67] on top of `angr` [230] which are dynamic symbolic execution tools and might have an additional support for concretization (but do not use it).

Four analyzers at binary level, prior to this work, constitute the state of the art [253, 71, 129, 67] to detect Spectre-PHT vulnerabilities but only two scale [253, 67]—by giving up on the precision (false positive). oo7 [253] relies on detecting vulnerable code pattern, whereas Pitchfork [67] relies on symbolic execution and taint analysis to detect secret dependent conditional statements and memory accesses.

The only previous work which addresses Spectre-STL is Pitchfork [67]. We have tested Pitchfork on our new Spectre-STL litmus tests for comparison with our work (cf. Table 6.5). We note that, although it is not documented [67], Pitchfork implements an

| Tool | Technique | Target | Property | Precise |
|---|---|---|---|---|
| AISE [261] | Abstract Interp. | LLVM | Cache | ✗ |
| KLEESpectre [252] | SE (KLEE) | LLVM | Cache | ✓ |
| SPECUSYM [132] | SE (KLEE) | LLVM | Cache | ✓ |
| oo7 [253] | Tainting | **Binary** | Patterns | ✗ |
| FASS [71] | MC (UCLID5) | **Binary** | SNI | ✓ |
| SPECTECTOR [129] | SE | **Binary** | SNI | ✓ |
| Pitchfork [67] | SE&taint. (`angr`) | **Binary** | SCT | ✗ |
| BINSEC/HAUNTED | RelSE (BINSEC) | **Binary** | SCT | ✓ |

| Tool | PHT | STL | Scales | Benchs |
|---|---|---|---|---|
| AISE [261] | ✓ NA | ✗ | ✓ | Crypto |
| KLEESpectre [252] | ✓ Explicit* | ✗ | ✓ | Crypto |
| SPECUSYM [132] | ✓ Explicit* | ✗ | ✓ | Crypto |
| oo7 [253] | ∼ NA | ✗ | ✓ | Other |
| FASS [71] | ✓ Explicit* | ✗ | ✗ | Litmus |
| SPECTECTOR [129] | ✓ Explicit* | ✗ | ✗ | Litmus |
| Pitchfork [67] | ✓ Explicit+ | ✓ Explicit | ✓ PHT / ✗ STL | Crypto |
| BINSEC/HAUNTED | ✓ Haunted | ✓ Haunted | ✓ PHT / ∼ STL | Crypto |

TABLE 6.6 – Comparison of BINSEC/HAUNTED with related work where SNI denotes speculative non-interference (transient executions do not leak more information than sequential executions). *These tools restrict to leaks in transient execution, so Haunted-PHT optimization does not apply, however their straightforward adaptation to SCT would be Explicit. +With optimizations.

optimized exploration technique compared to Explicit for Spectre-PHT. For Spectre-STL however, it relies on Explicit and forks the execution for each transient load. Therefore, it suffers from a significant state explosion problem for Spectre-STL and quickly runs out of memory.

Currently, there is no static analyzers addressing Spectre-BTB (speculative indirect branches) or Spectre-RSB (speculative returns). Although explicitly modeling transient paths underlying Spectre-BTB is in principle feasible, this is in practice intractable as it allows to jump to arbitrary addresses in the code on indirect jump instructions [67]. The same applies to Spectre-RSB, on recent Intel processors, when the return stack buffer is empty [158, 175].

**State merging in symbolic execution.** State merging [135, 120] (a.k.a. path merging) is used in symbolic execution to merge states following different paths (e.g. merge diverging paths after a conditional statement). Merging in symbolic execution precisely captures the behavior of the merged states, without over-approximation: the formula of the final state the disjunction of the formula of the state to be merged. While state merging reduces the number of paths to explore, it also increases the complexity of the formula [135], consequently techniques have been proposed to selectively apply state merging [160].

For the comparison, we adopt a different strategy: we do not pack together different paths encountered along the execution, but rather prevent creating artificial path

splits (unlike *Explicit*) by showing how to reason on both sequential and transient executions at the same time. In our setting, a path predicate represents all input values that follow a control-flow path, be it through sequential or transient executions. For Spectre-PHT this is achieved through a careful handling of assertions along symbolic execution (akin to pruning), whereas for STL this is achieved through a symbolic encoding of memory speculations inside the path predicate (somehow akin to some merge encodings, e.g. [160], for its use of *if-then-else* expressions).

## 6.8 Conclusion

We propose *Haunted RelSE*, a technique built on top of relational symbolic execution to statically detect Spectre-PHT and Spectre-STL vulnerabilities. Especially, Haunted RelSE allows to significantly alleviate the cost of addressing speculative paths by reasoning about sequential and transient executions at the same time.

We implement Haunted RelSE in a symbolic execution tool, Binsec/Haunted. Our experimental results show that Haunted RelSE is a step toward scalable analysis of Spectre attacks. For Spectre-PHT, Haunted RelSE can dramatically speed up the analysis in some cases, pruning the complexity of analyzing speculative semantics on medium size real world cryptographic binaries. For Spectre-STL, Binsec/Haunted is the first tool able to exhaustively analyze small real world cryptographic binaries and find vulnerabilities in medium size real world cryptographic binaries.

Finally, we report thanks to Binsec/Haunted that one standard defense for Spectre-PHT can easily introduce Spectre-STL vulnerabilities and propose a mitigation; and also that a well-known `gcc` option to compile to position independent executables introduces Spectre-STL vulnerabilities.

# Part IV

# Conclusion and Future Work

# Chapter 7

# Conclusion and Future Work

## 7.1 Conclusion

In this thesis, we address the problem of automating the security analysis of cryptographic implementations. Such analyses are essential because cryptographic primitives are pervasive and security-critical, whereas their desired properties are subtle and better checked at binary-level. We propose new binary-level symbolic analyses encompassing a subset of information flow policies restricted to pairs of traces following the same path. This subset includes crucial properties of cryptographic implementations such as constant-time, secret-erasure or speculative constant-time. Our technical contributions include two sets of optimizations, which make our analyses scalable:

— First, we propose dedicated optimizations for relational symbolic execution at binary level, called *Binary-level RelSE*. RelSE does not scale at binary-level because of the explicit representation of the memory as a symbolic array. To address this problem, our key technical insight is to improve sharing between pairs of execution in the symbolic memory, allowing for fine-grained secret tracking;

— Second, we propose dedicated optimizations to efficiently model the speculative semantics of programs and detect Spectre attacks, called *Haunted RelSE*. Modeling the speculative semantics of programs by representing transient executions explicitly quickly leads to path explosion. To alleviate the cost of modeling the speculative semantics, our key technical insight is to model transient executions at the same time as sequential execution.

We implement these optimizations into two open-source tools: Binsec/Rel for analyzing constant-time and secret-erasure, and Binsec/Haunted for analyzing speculative constant-time. Using our tools, we analyze cryptographic primitives taken from open-source cryptographic libraries such as Libsodium [41], OpenSSL [192], BearSSL [36], and HACL* [272]. Our experimental evaluation against prior techniques and state-of-the-art tools shows that our optimizations are crucial to scale on real-world cryptographic code, finding more bugs than prior approaches and performing bounded-verification when they timeout.

The properties that we target have in common that they are generally not preserved by compilers. In this context, reasoning at binary-level gives two main advantages. First, binary-analysis enables to find vulnerabilities introduced by compilers in binaries compiled from secure source code, and to automatically study the preservation of properties by compilers. This allows us to find vulnerabilities introduced by backend passes of `clang` which were out of reach of LLVM verification tools. Second, binary-level analysis is free from any assumption on the compiler; therefore and unlike source code analysis, our analysis does not require to be backed-up by a constant-time preserving compiler [35, 31]

Essentially, our work shows that, with appropriate optimizations, symbolic security analysis of cryptographic code scales at binary-level—even when considering speculative execution.

## 7.2 Perspectives

In this section, we detail some perspective for future work, building on the work presented so far.

**Frameworks for checking property preservation by compilers.** A particularly interesting application of the work presented in this thesis is the automatic analysis of countermeasures in multiple compilation setups to ensure that countermeasures are not optimized-away by compilers. Prior to this work, these kind of analyses were conducted manually [231, 269]. In this thesis, we demonstrate that it is possible to automate them at a large scale on constant-time and secret-erasure (respectively 408 and 680 programs). We believe that this line of work can be extended in several directions to systematically check countermeasures in multiple compilation setups (using multiple compilers and compilation options):

1. The analysis could be extended to other countermeasures such as software-based Spectre mitigations given in Section 3.3.4 (e.g. index-masking, speculative load hardening, or serializing instructions);
2. The analysis could be extended to new compilers and compiler options such as Microsoft's MSVC compiler and its `/Qspectre` flag for inserting defenses against Spectre-PHT, as done by Guarnieri et al. with their tool Spectector [129].

This could allow developers to design new enforcement mechanisms and easily test, in many compilation settings, that they are not optimized away.

**Automatic repair of binary code.** This thesis addresses the automatic discovery of Spectre vulnerabilities but does not address how to patch them. The current implementation of BINSEC/HAUNTED reports vulnerable instructions (where the leak happens), but does not provide feedback on how to patch the code (e.g. where to insert `fences`). Developers are expected to understand the vulnerability, come up with a patch, and manually apply it. However, coming up with a patch is not trivial, even when the address of the leaky instruction is known, which makes manual patching error prone. For instance, it is not sufficient to add an `lfence` instruction just before leaky instructions because `lfence` does not stop instructions pre-fetching, which can also leak secrets [225].

A solution could be to provide specific feedback on where to insert countermeasures in the assembly code. Even better, the assembly code could be automatically patched—as initiated by Blade [250] on WebAssembly. For instance `fence` instructions could be automatically inserted in the assembly code before recompiling the assembly code to binary.

If source (or assembly) code is not available, it is also possible to directly patch the binary code by modifying the disassembled code (on its intermediate representation) before recompiling it. This approach called *binary recompilation* [258] is already implemented in a tool called Egalito [258] and successfully applied for patching binary code against Spectre-BTB, replacing indirect jump instructions with retpoline sequences [1] [245, 215].

1. Retpoline is a countermeasure for Spectre-BTB that replaces indirect jumps with semantically equivalent sequences of instructions, using `call`/`ret`.

**Other Spectre variants.** In this thesis, we address two variants of Spectre—namely Spectre-PHT and Spectre-STL—and consider the two other variants—namely Spectre-BTB [155] and Spectre-RSB [175, 158]—as out of scope. Spectre-BTB exploits the indirect jump predictor which records possible targets of jump instructions in order to predict the outcome of subsequent indirect jumps. In principle the predictor can be trained by an attacker to jump to an arbitrary target.

Spectre-RSB exploits a microarchitectural component called the return stack buffer (RSB) which predict the target of return instructions. Contrary to the BTB, the behavior of the RSB is more predictable: at each `call` instruction, the RSB pushes the return target (i.e. address of the `call` + 1) to a stack, and at each `ret` instruction, it predicts the next target by popping the last entry from the stack. When the stack buffer is empty, return prediction either falls back to the BTB mechanism or cycle through the RSB. The RSB is not always shared between an attacker and its victim (e.g. it can be sanitized between context switches using RSB stuffing [215], or partitioned). Therefore, we can work with the assumption that the victim has $n$ entries in the return stack buffer that cannot be evicted by the attacker. Under this assumption, we can model a RSB with $n$ entries and consider that speculations at return instructions are determined by these entries. Finally, using this RSB we can:

1. Detect (insecure) `ret` instructions that operate on an empty stack buffer;
2. Formally reason about the main countermeasure for Spectre-BTB called Retpolines [245, 215] which consists in replacing indirect jumps with equivalent sequences of instructions containing `call` and `ret` instructions;
3. Detect and reason about mispredictions resulting from abnormal control flow—when a `ret` instruction does not return to its caller.

It would also be interesting to consider another countermeasure against Spectre-BTB—implemented in the Linux kernel—that leverages code patching in order to replace indirect calls with conditional direct calls [13, 206]. Analyzing this countermeasure with Binsec/Haunted would require to add support for rewriting code in Binsec.

Finally, we could adapt Haunted RelSE and Binsec/Haunted to support reasoning about the new predictor recently introduced in new ARM processors, called Predictive Store Forwarding (PSF) [2]. On top of speculatively bypassing `store` instructions when it speculates that a `load` *does not alias* with prior `store` instructions (Spectre-STL), processors implementing PSF can speculate that a `load` *does alias* with a `store` and speculatively forward the value from the `store` to the `load`.

**Exploitability.** It is currently unclear whether *constant-time* and *speculative constant-time* violations are actually exploitable to mount an attack. An interesting line of work would be to investigate the exploitability of the violations found by our tools. Several directions are possible there:

1. A first direction could be to quantify information leaks in order to give a bound on the number of secret bits that an attacker can extract [162, 232, 22, 164, 203, 204], or to propose low inputs maximizing the amount of leakage [202, 198];
2. A second direction would be to distinguish inputs that are controlled by an attacker from uncontrolled inputs (initial memory, or initial value of `esp`) and adopt a form of robust reachability [117]. A vulnerability is robustly reachable

2. Interestingly, this predictor has been formally defined in the speculative semantics of Cauligi et al. [67] and Guanciale, Balliu, and Dam [127], and the corresponding Spectre variant theorized before being actually implemented in processors.

if it can be triggered regardless of the value of the uncontrolled input, meaning that the attacker doesn't have to rely on uncontrolled inputs;

3. Finally, we could also investigate less conservative definitions of speculative constant-time, distinguishing *likely* speculations from *unlikely* speculations. For instance in the code `store [ebp-4] eax; load [ebp-4] eax`, speculative constant-time considers that the `store` instruction could be bypassed by the `load`. However, because indexes are syntactically equal, this is very unlikely to happen.

**Other directions.**

— Currently our analyses restrict to properties relating pairs of traces following the same path. This includes properties that forbid secret-dependent control flow (such as *constant-time*), properties that declassify control flow [219], or properties that are concerned with explicit flows and ignore implicit flows (such as *explicit secrecy* [222]). Accounting for the general *noninterference* policy would require to model pairs of traces following different paths with appropriate heuristics to handle path explosion;

— One of the main bottlenecks of BINSEC/HAUNTED and BINSEC/REL is the generation of counterexamples on insecure cryptographic codes. Indeed, BINSEC/REL spares *unsatisfiable* insecurity queries and is efficient on secure programs; however, it does not spare *satisfiable* insecurity queries and is less efficient on insecure codes (e.g. see `aes-big` in Table 4.4). On these programs, large and complex insecurity queries are sent to the solver to generate a model and report a counterexample; and the solver struggles to solve these queries. A solution would be to propagate concrete secret inputs along symbolic execution to directly detect differences in control-flow/memory accesses and report counterexamples;

— Our tools currently deal with loops by unrolling them. While it still allows us to verify programs with fixed-length inputs—such as `tea` or `donna`—, we cannot offer the same guarantees on programs with unbounded-length inputs—such as stream ciphers like `salsa20` or `chacha20`—, for which we only offer guarantees for a given input length[3]. To be able to deal with unbounded-length input a possible solution would be to use relational loop invariants [24]—however, it would sacrifice bug-finding;

— Finally, building on recent work by Guarnieri et al. [130], we could parameterize BINSEC/HAUNTED with a notion of hardware-software contracts. Contracts define the *observations* that an attacker can make as well as the *execution modes* (e.g. speculative semantics vs. sequential semantics), and programs are checked for noninterference with respect to these contracts. BINSEC/REL can already be parameterized by the leakage model rather easily—which accounts for the *observation* part of contracts. However, building tools that can be parameterized by the execution mode remains an open question.

3. While in practice we fix the input length, note that in theory we could keep it symbolic and guarantee correctness up to a given bound (with additional assumptions on the symbolic length to avoid buffer overflows).

# Bibliography


[1] *6.47.2 Extended Asm - Assembler Instructions with C Expression Operands*. URL: https://gcc.gnu.org/onlinedocs/gcc/Extended-Asm.html (visited on 04/24/2021) (cited on page 104).

[2] Onur Aciiçmez. « Yet Another MicroArchitectural Attack: : Exploiting i-Cache ». In: *CSAW*. 2007 (cited on pages 4, 7, 40).

[3] Onur Aciiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. « Predicting Secret Keys via Branch Prediction ». In: *CT-RSA*. 2007 (cited on pages 4, 7, 40, 54).

[4] Johan Agat. « Transforming out Timing Leaks ». In: *POPL* (Boston, MA, USA). 2000 (cited on page 84).

[5] Alejandro Cabrera Aldaya, Billy Bob Brumley, Sohaib ul Hassan, Cesar Pereida García, and Nicola Tuveri. « Port Contention for Fun and Profit ». In: *IEEE Symposium on Security and Privacy*. 2019 (cited on pages 4, 7, 40, 54).

[6] Nadhem J. AlFardan and Kenneth G. Paterson. « Lucky Thirteen: Breaking the TLS and DTLS Record Protocols ». In: *IEEE Symposium on Security and Privacy*. 2013 (cited on pages 56, 75–77, 191).

[7] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Arthur Blot, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Hugo Pacheco, Benedikt Schmidt, and Pierre-Yves Strub. « Jasmin: High-assurance and High-Speed Cryptography ». In: *CCS*. 2017 (cited on pages 6, 54).

[8] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, and François Dupressoir. « Verifiable Side-Channel Security of Cryptographic Implementations: Constant-Time MEE-CBC ». In: *Fast Software Encryption*. 2016 (cited on page 149).

[9] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, and François Dupressoir. *Verifying Constant-Time Implementations, benchmark [8, 10]*. URL: https://github.com/imdea-software/verifying-constant-time (visited on 07/12/2021) (cited on pages 76, 131).

[10] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, and Michael Emmi. « Verifying Constant-Time Implementations ». In: *USENIX Security Symposium*. 2016 (cited on pages 54, 56, 75, 77, 79, 83, 84, 149, 191).

[11] Bowen Alpern and Fred B. Schneider. « Defining Liveness ». In: *Inf. Process. Lett.* 21.4 (1985) (cited on page 35).

[12] Bowen Alpern and Fred B. Schneider. « Recognizing Safety and Liveness ». In: *Distributed Comput.* 2.3 (1987) (cited on pages 5, 35).

[13] Nadav Amit, Fred Jacobs, and Michael Wei. « JumpSwitches: Restoring the Performance of Indirect Branches in the Era of Spectre ». In: *USENIX Annual Technical Conference*. 2019 (cited on page 147).

[14] Marc Andrysco, David Kohlbrenner, Keaton Mowery, Ranjit Jhala, Sorin Lerner, and Hovav Shacham. « On Subnormal Floating Point and Abnormal Timing ». In: *IEEE Symposium on Security and Privacy*. 2015 (cited on pages 4, 40).

[15] Andrew W. Appel, Robert Dockins, Aquinas Hobor, Lennart Beringer, Josiah Dodds, Gordon Stewart, Sandrine Blazy, and Xavier Leroy. « The CompCert Memory Model ». In: *Program Logics for Certified Compilers*. 2014 (cited on page 25).

[16] *ArraysEx Theory, SMT-LIB*. URL: http://smtlib.cs.uiowa.edu/theories-ArraysEx.shtml (visited on 04/02/2019) (cited on page 28).

[17] Aslan Askarov, Scott Moore, Christos Dimoulas, and Stephen Chong. « Cryptographic Enforcement of Language-Based Information Erasure ». In: *CSF*. 2015 (cited on page 106).

[18] Thomas H. Austin and Cormac Flanagan. « Multiple Facets for Dynamic Information Flow ». In: *POPL*. 2012 (cited on pages 7, 37).

[19] Thanassis Avgerinos, David Brumley, John Davis, Ryan Goulden, Tyler Nighswander, Alexandre Rebert, and Ned Williamson. « The Mayhem Cyber Reasoning System ». In: *IEEE Security & Privacy* 16.2 (2018) (cited on pages 5, 20, 27, 54).

[20] Thanassis Avgerinos, Sang Kil Cha, Brent Lim Tze Hao, and David Brumley. « AEG: Automatic Exploit Generation ». In: *NDSS*. 2011 (cited on page 20).

[21] J. Bacelar Almeida, Manuel Barbosa, Jorge S. Pinto, and Bárbara Vieira. « Formal Verification of Side-Channel Countermeasures Using Self-Composition ». In: *Science of Computer Programming* 78.7 (2013) (cited on pages 54, 84).

[22] Michael Backes, Boris Köpf, and Andrey Rybalchenko. « Automatic Discovery and Quantification of Information Leaks ». In: *IEEE Symposium on Security and Privacy*. 2009 (cited on page 147).

[23] Gogul Balakrishnan and Thomas W. Reps. « WYSINWYX: What You See Is Not What You eXecute ». In: *ACM Transactions on Programming Languages and Systems* 32.6 (2010) (cited on pages 7, 22, 54).

[24] Musard Balliu, Mads Dam, and Roberto Guanciale. « Automating Information Flow Analysis of Low Level Code ». In: *CCS*. 2014 (cited on pages 6, 55, 83, 148).

[25] Musard Balliu, Mads Dam, and Gurvan Le Guernic. « ENCoVer: Symbolic Exploration for Information Flow Security ». In: *CSF*. 2012 (cited on page 83).

[26] Kristin Barber, Anys Bacha, Li Zhou, Yinqian Zhang, and Radu Teodorescu. « SpecShield: Shielding Speculative Data from Microarchitectural Covert Channels ». In: *PACT*. 2019 (cited on page 110).

[27] Sébastien Bardin, Robin David, and Jean-Yves Marion. « Backward-Bounded DSE: Targeting Infeasibility Questions on Obfuscated Codes ». In: *IEEE Symposium on Security and Privacy*. 2017 (cited on pages 20, 24, 27, 54, 82).

[28] Sébastien Bardin, Philippe Herrmann, Jérôme Leroux, Olivier Ly, Renaud Tabary, and Aymeric Vincent. « The BINCOA Framework for Binary Code Analysis ». In: *CAV*. 2011 (cited on pages 24, 25, 60, 73, 90).

[29] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. *The SMT-LIB Standard: Version 2.6*. 2017 (cited on pages 28, 62, 73, 95, 113).

[30] Gilles Barthe, Gustavo Betarte, Juan Diego Campo, Carlos Daniel Luna, and David Pichardie. « System-Level Non-Interference for Constant-Time Cryptography ». In: *CCS*. 2014 (cited on pages 4, 7, 41, 54, 84).

[31] Gilles Barthe, Sandrine Blazy, Benjamin Grégoire, Rémi Hutin, Vincent Laporte, David Pichardie, and Alix Trieu. « Formal Verification of a Constant-Time Preserving C Compiler ». In: *Proc. ACM Program. Lang.* 4 (POPL 2020) (cited on pages 6, 145).

[32] Gilles Barthe, Juan Manuel Crespo, and César Kunz. « Relational Verification Using Product Programs ». In: *FM*. 2011 (cited on page 37).

[33] Gilles Barthe, Pedro R. D'Argenio, and Tamara Rezk. « Secure Information Flow by Self-Composition ». In: *CSFW*. 2004 (cited on pages 6, 36, 54).

[34] Gilles Barthe, Benjamin Grégoire, and Santiago Zanella Béguelin. « Formal Certification of Code-Based Cryptographic Proofs ». In: *POPL*. 2009 (cited on page 37).

[35] Gilles Barthe, Benjamin Grégoire, and Vincent Laporte. « Secure Compilation of Side-Channel Countermeasures: The Case of Cryptographic "Constant-Time" ». In: *CSF*. 2018 (cited on pages 7, 60, 90, 91, 110, 138, 145).

[36] *BearSSL - Constant-Time Crypto*. URL: `https://bearssl.org/constanttime.html` (visited on 05/07/2019) (cited on pages 4, 41, 54, 56, 91, 145).

[37] D Elliott Bell and Leonard J La Padula. *Secure Computer System: Unified Exposition and Multics Interpretation*. 1976 (cited on page 34).

[38] Nick Benton. « Simple Relational Correctness Proofs for Static Analyses and Program Transformations ». In: *POPL*. 2004 (cited on pages 7, 37).

[39] Daniel J Bernstein. *Cache-Timing Attacks on AES*. 2005 (cited on pages 4, 7, 40, 54).

[40] Daniel J. Bernstein. « Curve25519: New Diffie-Hellman Speed Records ». In: *Public Key Cryptography - PKC 2006*. 2006 (cited on pages 8, 59, 75).

[41] Daniel J. Bernstein, Tanja Lange, and Peter Schwabe. « The Security Impact of a New Cryptographic Library ». In: *LATINCRYPT*. 2012 (cited on pages 4, 9, 41, 54, 56, 75, 76, 145, 159).

[42] Frédéric Besson, Sandrine Blazy, and Pierre Wilke. « A Precise and Abstract Memory Model for C Using Symbolic Values ». In: *APLAS*. 2014 (cited on page 25).

[43] Frédéric Besson, Alexandre Dang, and Thomas P. Jensen. « Information-Flow Preservation in Compiler Optimisations ». In: *CSF*. 2019 (cited on pages 5, 6, 41, 87, 89, 107).

[44] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Ofer Strichman, and Yunshan Zhu. « Bounded Model Checking ». In: *Adv. Comput.* 58 (2003) (cited on page 5).

[45] Sandrine Blazy, David Pichardie, and Alix Trieu. « Verifying Constant-Time Implementations by Abstract Interpretation ». In: *ESORICS (1)*. 2017 (cited on pages 54, 75, 79, 84).

[46] Barry Bond, Chris Hawblitzel, Manos Kapritsos, K. Rustan M. Leino, Jacob R. Lorch, Bryan Parno, Ashay Rane, Srinath T. V. Setty, and Laure Thompson. « Vale: Verifying High-Performance Cryptographic Assembly Code ». In: *USENIX Security Symposium*. 2017 (cited on page 54).

[47] Peter Boonstoppel, Cristian Cadar, and Dawson R. Engler. « RWset: Attacking Path Explosion in Constraint-Based Test Generation ». In: *TACAS*. 2008 (cited on page 21).

[48] Ella Bounimova, Patrice Godefroid, and David A. Molnar. « Billions and Billions of Constraints: Whitebox Fuzz Testing in Production ». In: *ICSE*. 2013 (cited on pages 6, 17, 20).

[49] Robert Brotzman, Shen Liu, Danfeng Zhang, Gang Tan, and Mahmut Kandemir. « CaSym: Cache Aware Symbolic Execution for Side Channel Detection and Mitigation ». In: *IEEE Symposium on Security and Privacy*. 2019 (cited on pages 6, 54, 55, 83, 84, 139).

[50] Billy Bob Brumley and Nicola Tuveri. « Remote Timing Attacks Are Still Practical ». In: *ESORICS*. 2011 (cited on pages 4, 53).

[51] David Brumley and Dan Boneh. « Remote Timing Attacks Are Practical ». In: *USENIX Security Symposium*. 2003 (cited on pages 4, 53).

[52] David Brumley and Dan Boneh. « Remote Timing Attacks Are Practical ». In: *Comput. Networks* 48.5 (2005) (cited on pages 4, 37).

[53] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J. Schwartz. « BAP: A Binary Analysis Platform ». In: *CAV*. 2011 (cited on page 22).

[54] *Bug 15495 - dead store pass ignores memory clobbering asm statement*. URL: https://bugs.llvm.org/show_bug.cgi?id=15495 (visited on 04/24/2021) (cited on pages 104, 105).

[55] Jo Van Bulck, Daniel Moghimi, Michael Schwarz, Moritz Lipp, Marina Minkin, Daniel Genkin, Yuval Yarom, Berk Sunar, Daniel Gruss, and Frank Piessens. « LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection ». In: *IEEE Symposium on Security and Privacy*. 2020 (cited on page 46).

[56] Yuriy Bulygin. « CPU Side-Channels vs. Virtualization Malware: The Good, the Bad or the Ugly ». In: *ToorCon: Seattle, Seattle, WA, US* (2008) (cited on page 40).

[57] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. « Directed Greybox Fuzzing ». In: *CCS*. 2017 (cited on page 17).

[58] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. « KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs ». In: *OSDI*. 2008 (cited on pages 5, 18, 20, 21, 83, 133, 139).

[59] Cristian Cadar and Dawson R. Engler. « Execution Generated Test Cases: How to Make Systems Code Crash Itself ». In: *SPIN*. 2005 (cited on page 18).

[60] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. « EXE: Automatically Generating Inputs of Death ». In: *ACM Transactions on Information and System Security* 12.2 (2008) (cited on page 20).

[61] Cristian Cadar and Martin Nowack. « KLEE Symbolic Execution Engine in 2019 ». In: *International Journal on Software Tools for Technology Transfer* (2020) (cited on page 20).

[62] Cristian Cadar and Hristina Palikareva. « Shadow Symbolic Execution for Better Testing of Evolving Software ». In: *ICSE Companion*. 2014 (cited on pages 7, 55, 58, 83).

[63] Cristian Cadar and Koushik Sen. « Symbolic Execution for Software Testing: Three Decades Later ». In: *Communications of the ACM* 56.2 (2013) (cited on pages 6, 18, 20, 54, 72).

[64] Giovanni Camurati, Sebastian Poeplau, Marius Muench, Tom Hayes, and Aurélien Francillon. « Screaming Channels: When Electromagnetic Side Channels Meet Radio Transceivers ». In: *CCS*. 2018 (cited on page 3).

[65] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. « A Systematic Evaluation of Transient Execution Attacks and Defenses ». In: *USENIX Security Symposium*. 2019 (cited on pages 4, 43, 46, 110, 138).

[66] Chandler Carruth. *Speculative Load Hardening*. URL: https://developer.arm.com/support/arm-security-updates/speculative-processor-vulnerability/download-the-whitepaper (visited on 07/25/2021) (cited on page 47).

[67] Sunjay Cauligi, Craig Disselkoen, Klaus von Gleissenthall, Dean M. Tullsen, Deian Stefan, Tamara Rezk, and Gilles Barthe. « Constant-Time Foundations for the New Spectre Era ». In: *PLDI*. 2020 (cited on pages 5, 7, 9, 43, 49, 110–112, 116, 129, 130, 133–135, 138–140, 147, 155, 186, 189).

[68] Sunjay Cauligi, Gary Soeller, Fraser Brown, Brian Johannesmeyer, Yunlu Huang, Ranjit Jhala, and Deian Stefan. « FaCT: A Flexible, Constant-Time Programming Language ». In: *SecDev*. 2017 (cited on page 54).

[69] Sudipta Chattopadhyay, Moritz Beck, Ahmed Rezine, and Andreas Zeller. « Quantifying the Information Leak in Cache Attacks via Symbolic Execution ». In: *MEMOCODE*. 2017 (cited on pages 54, 84).

[70] Sudipta Chattopadhyay and Abhik Roychoudhury. « Symbolic Verification of Cache Side-Channel Freedom ». In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 37.11 (2018) (cited on page 84).

[71] Kevin Cheang, Cameron Rasmussen, Sanjit A. Seshia, and Pramod Subramanyan. « A Formal Approach to Secure Speculation ». In: *CSF*. 2019 (cited on pages 110, 139, 140).

[72] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. « The S2E Platform: Design, Implementation, and Applications ». In: *ACM Transactions on Computer Systems* 30.1 (2012) (cited on pages 27, 54).

[73] Stephen Chong and Andrew C. Myers. « End-to-End Enforcement of Erasure and Declassification ». In: *CSF*. 2008 (cited on pages 3, 106).

[74] Stephen Chong and Andrew C. Myers. « Language-Based Information Erasure ». In: *CSFW*. 2005 (cited on pages 87, 106).

[75] Edmund M. Clarke and Jeannette M. Wing. « Formal Methods: State of the Art and Future Directions ». In: *ACM Computing Surveys* 28.4 (1996) (cited on pages 5, 15).

[76] Michael R. Clarkson, Bernd Finkbeiner, Masoud Koleini, Kristopher K. Micinski, Markus N. Rabe, and César Sánchez. « Temporal Logics for Hyperproperties ». In: *International Conference on Principles of Security and Trust*. 2014 (cited on page 5).

[77] Michael R. Clarkson and Fred B. Schneider. « Hyperproperties ». In: *CSF*. 2008 (cited on pages 5, 35, 54).

[78] Michael R. Clarkson and Fred B. Schneider. « Hyperproperties ». In: *J. Comput. Secur.* 18.6 (2010) (cited on page 35).

[79] Peter Collingbourne, Cristian Cadar, and Paul H. J. Kelly. « Symbolic Testing of OpenCL Code ». In: *Haifa Verification Conference*. 2011 (cited on page 20).

[80] Nassim Corteggiani, Giovanni Camurati, and Aurélien Francillon. « Inception: System-wide Security Testing of Real-World Embedded Systems Software ». In: *USENIX Security Symposium*. 2018 (cited on page 20).

[81] Patrick Cousot and Radhia Cousot. « Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints ». In: *POPL*. 1977 (cited on page 5).

[82] Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. « The ASTREÉ Analyzer ». In: *ESOP*. 2005 (cited on pages 5, 17).

[83] *CWE-14: Compiler Removal of Code to Clear Buffers*. URL: https://cwe.mitre.org/data/definitions/14.html (visited on 09/29/2020) (cited on pages 6, 21, 87, 89, 107).

[84] Lesly-Ann Daniel. *A Framework to automatically check the preservation of secret-erasure by compilers*. URL: https://github.com/binsec/rel_bench/tree/main/src/secret-erasure (visited on 07/06/2021) (cited on pages 9, 88).

[85] Lesly-Ann Daniel. *Binsec/Rel: A Symbolic Binary Analyzer for Constant-Time and Secret-Erasure*. URL: https://github.com/binsec/Rel (visited on 06/28/2021) (cited on pages 8, 9).

[86] Lesly-Ann Daniel. *Binsec/Rel: A Symbolic Binary Analyzer to Detect Spectre Attacks*. URL: https://github.com/binsec/haunted (visited on 06/28/2021) (cited on pages 8, 9).

[87] Lesly-Ann Daniel, Sébastien Bardin, and Tamara Rezk. « Binsec/Rel: Efficient Relational Symbolic Execution for Constant-Time at Binary-Level ». In: *IEEE Symposium on Security and Privacy*. 2020 (cited on pages 10, 110, 139).

[88] Lesly-Ann Daniel, Sébastien Bardin, and Tamara Rezk. « Hunting the Haunter - Efficient Relational Symbolic Execution for Spectre with Haunted RelSE ». In: *NDSS*. 2021 (cited on page 10).

[89] Robin David, Sébastien Bardin, Josselin Feist, Laurent Mounier, Marie-Laure Potet, Thanh Dinh Ta, and Jean-Yves Marion. « Specification of Concretization and Symbolization Policies in Symbolic Execution ». In: *ISSTA*. 2016 (cited on pages 19, 82).

[90] Robin David, Sébastien Bardin, Thanh Dinh Ta, Laurent Mounier, Josselin Feist, Marie-Laure Potet, and Jean-Yves Marion. « BINSEC/SE: A Dynamic Symbolic Execution Toolkit for Binary-Level Analysis ». In: *SANER*. 2016 (cited on pages 24, 27, 54, 73, 82).

[91] Drew Davidson, Benjamin Moench, Thomas Ristenpart, and Somesh Jha. « FIE on Firmware: Finding Vulnerabilities in Embedded Systems Using Symbolic Execution ». In: *USENIX Security Symposium*. 2013 (cited on page 20).

[92] Leonardo Mendonça de Moura and Nikolaj Bjørner. « Z3: An Efficient SMT Solver ». In: *TACAS*. 2008 (cited on pages 58, 82).

[93] Dorothy E. Denning. « A Lattice Model of Secure Information Flow ». In: *Communications of the ACM* 19.5 (1976) (cited on page 34).

[94] Jean-François Dhem, François Koeune, Philippe-Alexandre Leroux, Patrick Mestré, Jean-Jacques Quisquater, and Jean-Louis Willems. « A Practical Implementation of the Timing Attack ». In: *CARDIS*. 1998 (cited on page 38).

[95] Craig Disselkoen. *Pitchfork's [67] modified set of Paul Kocher's litmus tests [154]*. URL: `https://www.schneier.com/sccd/TEA.C` (visited on 07/12/2021) (cited on pages 44, 130, 187).

[96] Adel Djoudi and Sébastien Bardin. « BINSEC: Binary Code Analysis with Low-Level Regions ». In: *TACAS*. 2015 (cited on pages 22, 24, 25, 54).

[97] Adel Djoudi, Sébastien Bardin, and Éric Goubault. « Recovering High-Level Conditions from Binary Programs ». In: *FM*. 2016 (cited on pages 7, 23, 54).

[98] Quoc Huy Do, Richard Bubel, and Reiner Hähnle. « Exploit Generation for Information Flow Leaks in Object-Oriented Programs ». In: *IFIP International Information Security Conference*. 2015 (cited on pages 6, 55, 83).

[99] Goran Doychev, Dominik Feld, Boris Köpf, Laurent Mauborgne, and Jan Reineke. « CacheAudit: A Tool for the Static Analysis of Cache Side Channels ». In: *USENIX Security Symposium*. 2013 (cited on pages 54, 75, 84).

[100] Goran Doychev and Boris Köpf. « Rigorous Analysis of Software Countermeasures against Cache Attacks ». In: *PLDI*. 2017 (cited on pages 54, 84).

[101] Vijay D'Silva, Mathias Payer, and Dawn Xiaodong Song. « The Correctness-Security Gap in Compiler Optimization ». In: *2015 IEEE Symposium on Security and Privacy Workshops, SPW 2015, San Jose, CA, USA, May 21-22, 2015*. 2015 (cited on pages 5, 6, 87, 89, 107).

[102] Marco Eilers, Peter Müller, and Samuel Hitz. « Modular Product Programs ». In: *Programming Languages and Systems*. 2018 (cited on page 37).

[103] Dmitry Evtyushkin, Dmitry V. Ponomarev, and Nael B. Abu-Ghazaleh. « Jump over ASLR: Attacking Branch Predictors to Bypass ASLR ». In: *MICRO*. 2016 (cited on pages 4, 7, 40, 54).

[104] Dmitry Evtyushkin, Ryan Riley, Nael B. Abu-Ghazaleh, and Dmitry Ponomarev. « BranchScope: A New Side-Channel Attack on Directional Branch Predictor ». In: *ASPLOS*. 2018 (cited on pages 4, 7, 40, 54).

[105] Gian Pietro Farina, Stephen Chong, and Marco Gaboardi. « Relational Symbolic Execution ». In: *PPDP*. 2019 (cited on pages 7, 37, 55, 58, 60, 83, 139).

[106] Benjamin Farinier, Sébastien Bardin, Richard Bonichon, and Marie-Laure Potet. « Model Generation for Quantified Formulas: A Taint-Based Approach ». In: *CAV (2)*. 2018 (cited on page 82).

[107] Benjamin Farinier, Robin David, Sébastien Bardin, and Matthieu Lemerre. « Arrays Made Simpler: An Efficient, Scalable and Thorough Preprocessing ». In: *LPAR*. 2018 (cited on pages 8, 21, 24, 66, 67, 73, 81, 82, 117, 123, 128).

[108] Josselin Feist, Laurent Mounier, Sébastien Bardin, Robin David, and Marie-Laure Potet. « Finding the Needle in the Heap: Combining Static Analysis and Dynamic Symbolic Execution to Trigger Use-after-Free ». In: *SSPREW@ACSAC*. 2016 (cited on page 24).

[109] Bernd Finkbeiner, Christopher Hahn, and Hazem Torfah. « Model Checking Quantitative Hyperproperties ». In: *CAV (1)*. 2018 (cited on page 5).

[110] Bernd Finkbeiner, Markus N. Rabe, and César Sánchez. « Algorithms for Model Checking HyperLTL and HyperCTL$^{\hat{*}}$ ». In: *CAV (1)*. 2015 (cited on page 5).

[111] *FixedSizeBitVectors Theory, SMT-LIB*. URL: http://smtlib.cs.uiowa.edu/theories-FixedSizeBitVectors.shtml (visited on 04/02/2019) (cited on page 28).

[112] Free Software Foundation. *Position Independent Code (GNU Compiler Collection (GCC) Internals)*. URL: https://gcc.gnu.org/onlinedocs/gccint/PIC.html (visited on 10/12/2020) (cited on pages 9, 111, 112).

[113] F.Pizlo. *What Spectre and Meltdown Mean for WebKit*. 2018. URL: https://webkit.org/blog/8048/what-spectre-and-meltdown-mean-for-webkit/ (visited on 07/17/2020) (cited on pages 46–48, 111, 129, 130, 136).

[114] Qian Ge, Yuval Yarom, Tom Chothia, and Gernot Heiser. « Time Protection: The Missing OS Abstraction ». In: *EuroSys* (Dresden, Germany). 2019 (cited on page 54).

[115] Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. « A Survey of Microarchitectural Timing Attacks and Countermeasures on Contemporary Hardware ». In: 8.1 (2018) (cited on pages 40, 54).

[116] Daniel Genkin, Adi Shamir, and Eran Tromer. « Acoustic Cryptanalysis ». In: *J. Cryptol.* 30.2 (2017) (cited on page 3).

[117] Guillaume Girol, Benjamin Farinier, and Sébastien Bardin. « Not All Bugs Are Created Equal, but Robust Reachability Can Tell the Difference ». In: *CAV (1)*. 2021 (cited on pages 24, 147).

[118] Patrice Godefroid. « Compositional Dynamic Test Generation ». In: *POPL*. 2007 (cited on page 20).

[119] Patrice Godefroid, Nils Klarlund, and Koushik Sen. « DART: Directed Automated Random Testing ». In: *PLDI*. 2005 (cited on pages 17, 20).

[120] Patrice Godefroid, Michael Y. Levin, and David A. Molnar. « Automated Whitebox Fuzz Testing ». In: *NDSS*. 2008 (cited on pages 111, 140).

[121] Patrice Godefroid, Michael Y. Levin, and David A. Molnar. « SAGE: Whitebox Fuzzing for Security Testing ». In: *Communications of the ACM* 55.3 (2012) (cited on pages 5, 6, 17, 18, 20, 27, 54, 110).

[122] Joseph A. Goguen and José Meseguer. « Security Policies and Security Models ». In: *IEEE Symposium on Security and Privacy*. 1982 (cited on page 34).

[123] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. « Translation Leak-aside Buffer: Defeating Cache Side-Channel Protections with TLB Attacks ». In: *USENIX Security Symposium*. 2018 (cited on pages 4, 7, 40).

[124] Ben Gras, Kaveh Razavi, Erik Bosman, Herbert Bos, and Cristiano Giuffrida. « ASLR on the Line: Practical Cache Attacks on the MMU ». In: *NDSS*. 2017 (cited on page 40).

[125] Johann Großschädl, Elisabeth Oswald, Dan Page, and Michael Tunstall. « Side-Channel Analysis of Cryptographic Software via Early-Terminating Multiplications ». In: *ICISC*. 2009 (cited on pages 4, 40).

[126] Daniel Gruss, Julian Lettner, Felix Schuster, Olga Ohrimenko, István Haller, and Manuel Costa. « Strong and Efficient Cache Side-Channel Protection Using Hardware Transactional Memory ». In: *USENIX Security Symposium*. 2017 (cited on page 54).

[127] Roberto Guanciale, Musard Balliu, and Mads Dam. « InSpectre: Breaking and Fixing Microarchitectural Vulnerabilities by Formal Analysis ». In: *CCS*. 2020 (cited on pages 139, 147).

[128] Roberto Guanciale, Hamed Nemati, Christoph Baumann, and Mads Dam. « Cache Storage Channels: Alias-driven Attacks and Verified Countermeasures ». In: *IEEE Symposium on Security and Privacy*. 2016 (cited on page 40).

[129] Marco Guarnieri, Boris Köpf, José F. Morales, Jan Reineke, and Andrés Sánchez. « Spectector: Principled Detection of Speculative Information Flows ». In: *IEEE Symposium on Security and Privacy*. 2020 (cited on pages 110, 139, 140, 146).

[130] Marco Guarnieri, Boris Köpf, Jan Reineke, and Pepe Vila. « Hardware-Software Contracts for Secure Speculation ». In: *IEEE Symposium on Security and Privacy*. 2021 (cited on page 148).

[131] David Gullasch, Endre Bangerter, and Stephan Krenn. « Cache Games - Bringing Access-Based Cache Attacks on AES to Practice ». In: *IEEE Symposium on Security and Privacy*. 2011 (cited on pages 4, 7, 39, 40).

[132] Shengjian Guo, Yueqi Chen, Peng Li, Yueqiang Cheng, Huibo Wang, Meng Wu, and Zhiqiang Zuo. « SpecuSym: Speculative Symbolic Execution for Cache Timing Leak Detection ». In: ICSE 2020 Technical Papers. 2020 (cited on pages 110, 121, 139, 140).

[133] *HACL*, `Lib_Memzero0_memzero` function*. URL: https://github.com/project-everest/hacl-star/blob/v0.3.0/lib/c/Lib_Memzero0.c (visited on 04/24/2021) (cited on page 104).

[134] René Rydhof Hansen and Christian W Probst. « Non-Interference and Erasure Policies for Java Card Bytecode ». In: *6th International Workshop on Issues in the Theory of Security (WITS'06)*. 2006 (cited on page 106).

[135] Trevor Hansen, Peter Schachte, and Harald Søndergaard. « State Joining and Splitting for the Symbolic Execution of Binaries ». In: *RV*. 2009 (cited on pages 111, 140).

[136] Klaus Havelund and Thomas Pressburger. « Model Checking JAVA Programs Using JAVA PathFinder ». In: *Int. J. Softw. Tools Technol. Transf.* 2.4 (2000) (cited on page 5).

[137] John Hazen. *Mitigating speculative execution side-channel attacks in Microsoft Edge and Internet Explorer*. 2018. URL: https://blogs.windows.com/msedgedev/2018/01/03/speculative-execution-mitigations-microsoft-edge-internet-explorer/ (visited on 07/25/2021) (cited on page 48).

[138] Shaobo He, Michael Emmi, and Gabriela F. Ciocarlie. « Ct-Fuzz: Fuzzing for Timing Leaks ». In: *ICST*. 2020 (cited on page 84).

[139] Jann Horn. *Speculative Execution, Variant 4: Speculative Store Bypass*. 2018. URL: https://bugs.chromium.org/p/project-zero/issues/detail?id=1528 (visited on 10/12/2020) (cited on pages 43, 45, 110).

[140] Sebastian Hunt and David Sands. « Just Forget It - the Semantics and Enforcement of Information Erasure ». In: *ESOP*. 2008 (cited on page 106).

[141] *Imdea-Software/Verifying-Constant-Time*. GitHub. URL: https://github.com/imdea-software/verifying-constant-time (visited on 10/13/2019) (cited on pages 75, 76).

[142] Intel. *Arm whitepaper – Cache Speculation Side-channels*. 2020. URL: https://developer.arm.com/support/arm-security-updates/speculative-processor-vulnerability/download-the-whitepaper (visited on 07/25/2021) (cited on page 46).

[143] Intel. *Speculative Execution Side Channel Mitigations*. 2018. URL: https://software.intel.com/content/dam/develop/external/us/en/documents/336996-speculative-execution-side-channel-mitigations.pdf (visited on 07/25/2021) (cited on pages 46, 48).

[144] *Intel® 64 and IA-32 Architectures Optimization Reference Manual*. Intel. URL: https://www.intel.com/content/www/us/en/develop/download/intel-64-and-ia-32-architectures-optimization-reference-manual.html (visited on 10/12/2020) (cited on pages 45, 123).

[145] Yeongjin Jang, Sangho Lee, and Taesoo Kim. « Breaking Kernel Address Space Layout Randomization with Intel TSX ». In: *CCS*. 2016 (cited on page 40).

[146] Jacques-Henri Jourdan, Vincent Laporte, Sandrine Blazy, Xavier Leroy, and David Pichardie. « A Formally-Verified C Static Analyzer ». In: *POPL*. 2015 (cited on page 5).

[147] Thierry Kaufmann, Hervé Pelletier, Serge Vaudenay, and Karine Villegas. « When Constant-Time Source Yields Variable-Time Binary: Exploiting Curve25519-Donna Built with MSVC 2015 ». In: *CANS*. 2016 (cited on page 54).

[148] Khaled N. Khasawneh, Esmaeil Mohammadian Koruyeh, Chengyu Song, Dmitry Evtyushkin, Dmitry Ponomarev, and Nael B. Abu-Ghazaleh. « SafeSpec: Banishing the Spectre of a Meltdown with Leakage-Free Speculation ». In: *DAC*. 2019 (cited on page 110).

[149] Soomin Kim, Markus Faerevaag, Minkyu Jung, SeungIl Jung, DongYeop Oh, JongHyup Lee, and Sang Kil Cha. « Testing Intermediate Representations for Binary Analysis ». In: *ASE*. 2017 (cited on pages 24, 82).

[150] Johannes Kinder. « Hypertesting: The Case for Automated Testing of Hyperproperties ». In: *3rd Workshop on Hot Issues in Security Principles and Trust (HotSpot)*. 2015 (cited on page 5).

[151] James C. King. « Symbolic Execution and Program Testing ». In: *Communications of the ACM* 19.7 (1976) (cited on pages 5, 18, 110).

[152] Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. « Frama-c: A Software Analysis Perspective ». In: *Formal Aspects Comput.* 27.3 (2015) (cited on pages 5, 17).

[153] Vladimir Kiriansky and Carl A. Waldspurger. « Speculative Buffer Overflows: Attacks and Defenses ». In: *CoRR* abs/1807.03757 (2018) (cited on pages 43, 138).

[154] Paul Kocher. *Spectre Mitigations in Microsoft's C/C++ Compiler*. 13, 2018. URL: https://www.paulkocher.com/doc/MicrosoftCompilerSpectreMitigation.html (visited on 07/30/2020) (cited on pages 46, 48, 110, 111, 129, 155, 187, 193).

[155] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. « Spectre Attacks: Exploiting Speculative Execution ». In: *IEEE Symposium on Security and Privacy*. 2019 (cited on pages 4, 43, 147).

[156] Paul C. Kocher. « Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems ». In: *CRYPTO*. 1996 (cited on pages 3, 37, 53).

[157] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. « Differential Power Analysis ». In: *CRYPTO*. 1999 (cited on pages 3, 37, 53).

[158] Esmaeil Mohammadian Koruyeh, Khaled N. Khasawneh, Chengyu Song, and Nael B. Abu-Ghazaleh. « Spectre Returns! Speculation Attacks Using the Return Stack Buffer ». In: *WOOT @ USENIX Security Symposium*. 2018 (cited on pages 43, 140, 147).

[159] Tomasz Kuchta, Hristina Palikareva, and Cristian Cadar. « Shadow Symbolic Execution for Testing Software Patches ». In: *ACM Trans. Softw. Eng. Methodol.* 27.3 (2018) (cited on pages 55, 58, 83).

[160] Volodymyr Kuznetsov, Johannes Kinder, Stefan Bucur, and George Candea. « Efficient State Merging in Symbolic Execution ». In: *PLDI*. 2012 (cited on pages 21, 111, 140, 141).

[161] Alfred Kölbl and Carl Pixley. « Constructing Efficient Formal Models from High-Level Descriptions Using Symbolic Simulation ». In: *Int. J. Parallel Program.* 33.6 (2005) (cited on page 21).

[162] Boris Köpf and David A. Basin. « An Information-Theoretic Model for Adaptive Side-Channel Attacks ». In: *CCS*. 2007 (cited on page 147).

[163] Boris Köpf and Heiko Mantel. « Transformational Typing and Unification for Automatically Correcting Insecure Programs ». In: *Int. J. Inf. Sec.* 6.2-3 (2007) (cited on page 84).

[164] Boris Köpf, Laurent Mauborgne, and Martín Ochoa. « Automatic Quantification of Cache Side-Channels ». In: *CAV*. 2012 (cited on pages 54, 84, 147).

[165] Leslie Lamport. « Proving the Correctness of Multiprocess Programs ». In: *IEEE Transactions on Software Engineering* 3.2 (1977) (cited on page 35).

[166] Adam Langley. *ImperialViolet - Checking That Functions Are Constant Time with Valgrind*. 2010. URL: https://www.imperialviolet.org/2010/04/01/ctgrind.html (visited on 03/14/2019) (cited on pages 54, 84).

[167] Adam Langley. *Implementations of a fast Elliptic-curve Diffie-Hellman primitive [41]*. URL: https://github.com/agl/curve25519-donna/tree/master (visited on 07/12/2021) (cited on pages 76, 131).

[168] Xavier Leroy. « Formal Verification of a Realistic Compiler ». In: *Communications of the ACM* 52.7 (2009) (cited on pages 6, 22, 107).

[169] *Libgcrypt, `wipememory` function*. URL: https://github.com/equalitie/libgcrypt/blob/libgcrypt-1.6.3/src/gcryptrnd.c (visited on 04/24/2021) (cited on page 103).

[170] *libsodium, `sodium_memzero` function*. URL: https://github.com/jedisct1/libsodium/blob/1.0.18/src/libsodium/sodium/utils.c (visited on 04/24/2021) (cited on pages 104, 105).

[171] Andreas Lindner, Roberto Guanciale, and Roberto Metere. « TrABin: Trustworthy Analyses of Binaries ». In: *Sci. Comput. Program.* 174 (2019) (cited on page 22).

[172] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. « Meltdown: Reading Kernel Memory from User Space ». In: *USENIX Security Symposium*. 2018 (cited on page 4).

[173] Fangfei Liu, Qian Ge, Yuval Yarom, Frank McKeen, Carlos V. Rozas, Gernot Heiser, and Ruby B. Lee. « CATalyst: Defeating Last-Level Cache Side Channel Attacks in Cloud Computing ». In: *HPCA*. 2016 (cited on page 54).

[174] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. « Last-Level Cache Side-Channel Attacks Are Practical ». In: *IEEE Symposium on Security and Privacy*. 2015 (cited on page 39).

[175] Giorgi Maisuradze and Christian Rossow. « Ret2spec: Speculative Execution Using Return Stack Buffers ». In: *CCS*. 2018 (cited on pages 43, 140, 147).

[176] *Managed Runtime Speculative Execution Side Channel Mitigations*. URL: https://software.intel.com/security-software-guidance/deep-dives/deep-dive-managed-runtime-speculative-execution-side-channel-mitigations (visited on 10/12/2020) (cited on page 110).

[177] Heiko Mantel, Alexandra Weber, and Boris Köpf. « A Systematic Study of Cache Side Channels across AES Implementations ». In: *ESSoS*. 2017 (cited on page 40).

[178] Paul Dan Marinescu and Cristian Cadar. « Make Test-Zesti: A Symbolic Execution Solution for Improving Regression Testing ». In: *ICSE*. 2012 (cited on page 20).

[179] Clémentine Maurice, Nicolas Le Scouarnec, Christoph Neumann, Olivier Heen, and Aurélien Francillon. « Reverse Engineering Intel Last-Level Cache Complex Addressing Using Performance Counters ». In: *RAID*. 2015 (cited on page 7).

[180] Xiaozhu Meng and Barton P. Miller. « Binary Code Is Not Easy ». In: *ISSTA*. 2016 (cited on page 23).

[181] Todd C. Miller. *sudo, `explicit_bzero` function*. URL: https://github.com/sudo-project/sudo/blob/SUDO_1_9_6/lib/util/explicit_bzero.c (visited on 04/24/2021) (cited on page 103).

[182] Dimiter Milushev, Wim Beck, and Dave Clarke. « Noninterference via Symbolic Execution ». In: *FMOODS/FORTE*. 2012 (cited on pages 6, 55, 83).

[183] David Molnar, Matt Piotrowski, David Schultz, and David A. Wagner. « The Program Counter Security Model: Automatic Detection and Removal of Control-Flow Side Channel Attacks ». In: *ICISC*. 2005 (cited on page 84).

[184] Aleksandar Nanevski, Anindya Banerjee, and Deepak Garg. « Verification of Information Flow and Access Control Policies with Dependent Types ». In: *IEEE Symposium on Security and Privacy*. 2011 (cited on page 106).

[185] Greg Nelson and Derek C. Oppen. « Simplification by Cooperating Decision Procedures ». In: *ACM Trans. Program. Lang. Syst.* 1.2 (1979) (cited on page 70).

[186] Nicholas Nethercote and Julian Seward. « Valgrind: A Framework for Heavy-weight Dynamic Binary Instrumentation ». In: *PLDI*. 2007 (cited on pages 84, 106).

[187] Minh Ngo, Nataliia Bielova, Cormac Flanagan, Tamara Rezk, Alejandro Russo, and Thomas Schmitz. « A Better Facet of Dynamic Information Flow Control ». In: *WWW (Companion Volume)*. 2018 (cited on page 37).

[188] Manh-Dung Nguyen, Sébastien Bardin, Richard Bonichon, Roland Groz, and Matthieu Lemerre. « Binary-Level Directed Fuzzing for Use-after-Free Vulnerabilities ». In: *RAID*. 2020 (cited on page 24).

[189] Olivier Nicole, Matthieu Lemerre, Sébastien Bardin, and Xavier Rival. « No Crash, No Exploit: Automated Verification of Embedded Kernels ». In: *RTAS*. 2021 (cited on page 24).

[190] Aina Niemetz and Mathias Preiner. « Bitwuzla at the SMT-COMP 2020 ». In: *CoRR* abs/2006.01621 (2020) (cited on pages 25, 82).

[191] Aina Niemetz, Mathias Preiner, and Armin Biere. « Boolector 2.0 System Description ». In: *Journal on Satisfiability, Boolean Modeling and Computation* 9 (2014 (published 2015)) (cited on pages 73, 128).

[192] *OpenSSL, Cryptography and SSL/TLS Toolkit*. URL: https://www.openssl.org/ (visited on 04/24/2021) (cited on pages 9, 75, 76, 78, 145).

[193] *OpenSSL, `OPENSSL_cleanse` function*. URL: https://github.com/openssl/openssl/blob/master/crypto/mem_clr.c (visited on 04/24/2021) (cited on page 102).

[194] Yossef Oren, Vasileios P. Kemerlis, Simha Sethumadhavan, and Angelos D. Keromytis. « The Spy in the Sandbox: Practical Cache Attacks in JavaScript and Their Implications ». In: *CCS*. 2015 (cited on page 40).

[195] Dag Arne Osvik, Adi Shamir, and Eran Tromer. « Cache Attacks and Countermeasures: The Case of AES ». In: *CT-RSA*. 2006 (cited on pages 4, 7, 39, 40, 54).

[196] Hristina Palikareva, Tomasz Kuchta, and Cristian Cadar. « Shadow of a Doubt: Testing for Divergences between Software Versions ». In: *ICSE*. 2016 (cited on pages 55, 58, 83).

[197] Andrew Pardoe. *Spectre mitigations in MSVC*. 2018. URL: https://devblogs.microsoft.com/cppblog/spectre-mitigations-in-msvc/ (visited on 07/25/2021) (cited on page 46).

[198] Corina S. Pasareanu, Quoc-Sang Phan, and Pasquale Malacaria. « Multi-Run Side-Channel Analysis Using Symbolic Execution and Max-Smt ». In: *CSF*. 2016 (cited on page 147).

[199] Colin Percival. *Cache Missing for Fun and Profit*. 2005 (cited on page 54).

[200] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. « DRAMA: Exploiting DRAM Addressing for Cross-Cpu Attacks ». In: *USENIX Security Symposium*. 2016 (cited on pages 4, 7, 40).

[201] Quoc-Sang Phan. « Self-Composition by Symbolic Execution ». In: *ICCSW*. 2013 (cited on page 57).

[202] Quoc-Sang Phan, Lucas Bang, Corina S. Pasareanu, Pasquale Malacaria, and Tevfik Bultan. « Synthesis of Adaptive Side-Channel Attacks ». In: *CSF*. 2017 (cited on page 147).

[203] Quoc-Sang Phan and Pasquale Malacaria. « Abstract Model Counting: A Novel Approach for Quantification of Information Leaks ». In: *AsiaCCS*. 2014 (cited on page 147).

[204] Quoc-Sang Phan, Pasquale Malacaria, Oksana Tkachuk, and Corina S. Pasareanu. « Symbolic Quantitative Information Flow ». In: *ACM SIGSOFT Softw. Eng. Notes* 37.6 (2012) (cited on page 147).

[205] Sebastian Poeplau and Aurélien Francillon. « Systematic Comparison of Symbolic Execution Systems: Intermediate Representation and Its Generation ». In: *ACSAC*. 2019 (cited on page 22).

[206] Josh Poimboeuf. *[PATCH v2 0/4] Static Calls [LWN.Net]*. 26, 2018. URL: https://lwn.net/ml/linux-kernel/cover.1543200841.git.jpoimboe@redhat.com/ (visited on 08/24/2021) (cited on page 147).

[207] Thomas Pornin. *BearSSL*. URL: https://www.bearssl.org/ (visited on 05/23/2019) (cited on pages 9, 75–77).

[208] Thomas Pornin. *Constant-time implementation of RSA in BearSSL*. URL: https://www.bearssl.org/constanttime.html#rsa (visited on 08/03/2021) (cited on page 41).

[209] Josyula R. Rao and Pankaj Rohatgi. « EMpowering Side-Channel Attacks ». In: *IACR Cryptol. ePrint Arch.* 2001 (2001) (cited on pages 3, 37, 53).

[210] Frédéric Recoules, Sébastien Bardin, Richard Bonichon, Matthieu Lemerre, Laurent Mounier, and Marie-Laure Potet. « Interface Compliance of Inline Assembly: Automatically Check, Patch and Refine ». In: *ICSE*. 2021 (cited on page 24).

[211] Frédéric Recoules, Sébastien Bardin, Richard Bonichon, Laurent Mounier, and Marie-Laure Potet. « Get Rid of Inline Assembly through Verification-Oriented Lifting ». In: *ASE*. 2019 (cited on pages 24, 82).

[212] Charles Reis, Alexander Moshchuk, and Nasko Oskov. « Site Isolation: Process Separation for Web Sites within the Browser ». In: *USENIX Security Symposium*. 2019 (cited on page 47).

[213] Xida Ren, Logan Moody, Mohammadkazem Taram, Matthew Jordan, Dean M Tullsen, and Ashish Venkat. « I See Dead $\mu$ops: Leaking Secrets via Intel/AMD Micro-Op Caches ». In: *ICSA* (2021) (cited on page 40).

[214] Oscar Reparaz, Josep Balasch, and Ingrid Verbauwhede. « Dude, Is My Code Constant Time? » In: *DATE*. 2017 (cited on page 84).

[215] *Retpoline: A Branch Target Injection Mitigation — Intel Whitepaper*. 2018. URL: https://software.intel.com/content/dam/develop/external/us/en/documents/retpoline-a-branch-target-injection-mitigation.pdf (visited on 08/17/2021) (cited on pages 146, 147).

[216] Bruno Rodrigues, Fernando Magno Quintão Pereira, and Diego F. Aranha. « Sparse Representation of Implicit Flows with Applications to Side-Channel Detection ». In: *CC*. 2016 (cited on page 84).

[217] Eyal Ronen, Kenneth G. Paterson, and Adi Shamir. « Pseudo Constant Time Implementations of TLS Are Only Pseudo Secure ». In: *CCS* (Toronto, Canada). 2018 (cited on pages 54, 191).

[218] Andrei Sabelfeld and Andrew C. Myers. « Language-Based Information-Flow Security ». In: *IEEE Journal on Selected Areas in Communications* 21.1 (2003) (cited on pages 5, 36).

[219] Andrei Sabelfeld and David Sands. « Declassification: Dimensions and Principles ». In: *J. Comput. Secur.* 17.5 (2009) (cited on page 148).

[220] Jonathan Salwan, Sébastien Bardin, and Marie-Laure Potet. « Symbolic Deobfuscation: From Virtualized Code Back to the Original ». In: *DIMVA*. 2018 (cited on pages 24, 27, 54).

[221] Prateek Saxena, Devdatta Akhawe, Steve Hanna, Feng Mao, Stephen McCamant, and Dawn Song. « A Symbolic Execution Framework for JavaScript ». In: *IEEE Symposium on Security and Privacy*. 2010 (cited on page 20).

[222] Daniel Schoepe, Musard Balliu, Benjamin C. Pierce, and Andrei Sabelfeld. « Explicit Secrecy: A Policy for Taint Tracking ». In: *EuroS&P*. 2016 (cited on page 148).

[223] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. « Q: Exploit Hardening Made Easy ». In: *USENIX Security Symposium*. 2011 (cited on page 20).

[224] Michael Schwarz, Moritz Lipp, and Daniel Gruss. « JavaScript Zero: Real JavaScript and Zero Side-Channel Attacks ». In: *NDSS*. 2018 (cited on page 48).

[225] Michael Schwarz, Martin Schwarzl, Moritz Lipp, Jon Masters, and Daniel Gruss. « NetSpectre: Read Arbitrary Memory over Network ». In: *ESORICS (1)*. 2019 (cited on pages 4, 7, 40, 46, 54, 146, 190).

[226] *Secretbox implementation in Libsodium []*. URL: https://github.com/jedisct1/libsodium/blob/master/src/libsodium/crypto_secretbox/crypto_secretbox.c (visited on 07/12/2021) (cited on page 131).

[227] *Secretgrind*. URL: https://github.com/lmrs2/secretgrind (visited on 09/29/2020) (cited on pages 87, 106).

[228] Koushik Sen, Darko Marinov, and Gul Agha. « CUTE: A Concolic Unit Testing Engine for C ». In: *ESEC/SIGSOFT FSE*. 2005 (cited on page 20).

[229] Ron Shemer, Arie Gurfinkel, Sharon Shoham, and Yakir Vizel. « Property Directed Self Composition ». In: *Computer Aided Verification*. 2019 (cited on page 37).

[230] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Krügel, and Giovanni Vigna. « SOK: (State of) the Art of War: Offensive Techniques in Binary Analysis ». In: *IEEE Symposium on Security and Privacy*. 2016 (cited on pages 22, 27, 54, 134, 139).

[231] Laurent Simon, David Chisnall, and Ross J. Anderson. « What You Get Is What You C: Controlling Side Effects in Mainstream C Compilers ». In: *EuroS&P*. 2018 (cited on pages 5, 6, 9, 41, 42, 54, 56, 75–79, 87, 107, 110, 139, 146).

[232] Geoffrey Smith. « On the Foundations of Quantitative Information Flow ». In: *FoSSaCS*. 2009 (cited on page 147).

[233] *SMT-COMP*. SMT-COMP. URL: https://smt-comp.github.io/2019/results.html (visited on 10/11/2019) (cited on pages 73, 128).

[234] *SMT-COMP 2019 – Competition Results for the ABVFP Division in the Single Query Track*. 2019. URL: https://smt-comp.github.io/2019/results/abvfp-single-query (cited on page 25).

[235] *SMT-COMP 2020 – Competition Results for the ABVFP Division in the Single Query Track*. 2020. URL: https://smt-comp.github.io/2020/results/qf-abvfp-single-query (cited on page 25).

[236] JaeSeung Song, Cristian Cadar, and Peter R. Pietzuch. « SymbexNet: Testing Network Protocol Implementations with Symbolic Execution and Rule-Based Specifications ». In: *IEEE Transactions on Software Engineering* 40.7 (2014) (cited on page 20).

[237] Marcelo Sousa and Isil Dillig. « Cartesian Hoare Logic for Verifying K-Safety Properties ». In: *PLDI*. 2016 (cited on page 37).

[238] Pramod Subramanyan, Sharad Malik, Hareesh Khattri, Abhranil Maiti, and Jason M. Fung. « Verifying Information Flow Properties of Firmware Using Symbolic Execution ». In: *DATE*. 2016 (cited on pages 6, 20, 55, 60, 83, 84, 139).

[239] Filippo Del Tedesco, Sebastian Hunt, and David Sands. « A Semantic Hierarchy for Erasure Policies ». In: *ICISS*. 2011 (cited on page 106).

[240] Filippo Del Tedesco, Alejandro Russo, and David Sands. « Implementing Erasure Policies Using Taint Analysis ». In: *NordSec*. 2010 (cited on page 106).

[241] Tachio Terauchi and Alexander Aiken. « Secure Information Flow as a Safety Problem ». In: *SAS*. 2005 (cited on pages 6, 37, 54).

[242] *The Chromium Project – Mitigating Side-Channel Attacks*. URL: https://www.chromium.org/Home/chromium-security/ssca (visited on 07/25/2021) (cited on page 48).

[243] *The GNU indirect function support (IFUNC)*. URL: https://sourceware.org/glibc/wiki/GNU_IFUNC (visited on 04/23/2021) (cited on page 183).

[244] Henrik Theiling. « Extracting Safe and Precise Control Flow from Binaries ». In: *RTCSA*. 2000 (cited on page 23).

[245] Paul Turner. *Retpoline: A Software Construct for Preventing Branch-Target-Injection*. URL: https://support.google.com/faqs/answer/7625886 (visited on 08/17/2021) (cited on pages 146, 147).

[246] Reini Urban. *Safeclib, `MEMORY_BARRIER` macro*. URL: https://github.com/rurban/safeclib/blob/v31082020/src/mem/mem_primitives_lib.h (visited on 04/24/2021) (cited on page 104).

[247] Reini Urban. *Safeclib, `memset_s` function*. URL: https://github.com/rurban/safeclib/blob/v31082020/src/mem/memset_s.c (visited on 04/24/2021) (cited on page 106).

[248] Stephan van Schaik, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. « Malicious Management Unit: Why Stopping Cache Attacks in Software Is Harder than You Think ». In: *USENIX Security Symposium*. 2018 (cited on page 39).

[249] Julien Vanegue and Sean Heelan. « SMT Solvers in Software Security ». In: *WOOT*. 2012 (cited on page 18).

[250] Marco Vassena, Craig Disselkoen, Klaus von Gleissenthall, Sunjay Cauligi, Rami Gökhan Kici, Ranjit Jhala, Dean M. Tullsen, and Deian Stefan. « Automatically Eliminating Speculative Leaks from Cryptographic Code with Blade ». In: *Proc. ACM Program. Lang.* 5 (POPL 2021) (cited on pages 46, 146).

[251] Luke Wagner. *Mitigations landing for new class of timing attack – Mozilla Security Blog*. 2018. URL: https://blog.mozilla.org/security/2018/01/03/mitigations-landing-new-class-timing-attack/ (visited on 07/25/2021) (cited on page 48).

[252] Guanhua Wang, Sudipta Chattopadhyay, Arnab Kumar Biswas, Tulika Mitra, and Abhik Roychoudhury. « KLEESpectre: Detecting Information Leakage through Speculative Cache Attacks via Symbolic Execution ». In: *ACM Trans. Softw. Eng. Methodol.* 29.3 (2020) (cited on pages 9, 110, 113, 114, 126, 129, 133, 139, 140, 189, 190).

[253] Guanhua Wang, Sudipta Chattopadhyay, Ivan Gotovchits, Tulika Mitra, and Abhik Roychoudhury. « Oo7: Low-overhead Defense against Spectre Attacks via Program Analysis ». In: *IEEE Transactions on Software Engineering* (2020) (cited on pages 46, 110, 139, 140).

[254] Shuai Wang, Pei Wang, Xiao Liu, Danfeng Zhang, and Dinghao Wu. « CacheD: Identifying Cache-Based Timing Channels in Production Software ». In: *USENIX Security Symposium*. 2017 (cited on pages 6, 54, 55, 83, 84, 139).

[255] David Wheeler and Roger Needham. *Implementation of The Tiny Encryption Algorithm (TEA) [256]*. URL: https://www.schneier.com/sccd/TEA.C (visited on 07/12/2021) (cited on pages 76, 131).

[256] David J. Wheeler and Roger M. Needham. « TEA, a Tiny Encryption Algorithm ». In: *Fast Software Encryption*. 1995 (cited on pages 75, 78, 165).

[257] Jan Wichelmann, Ahmad Moghimi, Thomas Eisenbarth, and Berk Sunar. « MicroWalk: A Framework for Finding Side Channels in Binaries ». In: *ACSAC* (San Juan, PR, USA). 2018 (cited on pages 54, 84).

[258] David Williams-King, Hidenori Kobayashi, Kent Williams-King, Graham Patterson, Frank Spano, Yu Jian Wu, Junfeng Yang, and Vasileios P. Kemerlis. « Egalito: Layout-agnostic Binary Recompilation ». In: *ASPLOS*. 2020 (cited on page 146).

[259] *wolfSSL, `ForceZero` function*. URL: https://github.com/equalitie/libgcrypt/blob/libgcrypt-1.6.3/src/gcryptrnd.c (visited on 04/24/2021) (cited on page 103).

[260] Meng Wu, Shengjian Guo, Patrick Schaumont, and Chao Wang. « Eliminating Timing Side-Channel Leaks Using Program Repair ». In: *ISSTA*. 2018 (cited on page 84).

[261] Meng Wu and Chao Wang. « Abstract Interpretation under Speculative Execution ». In: *PLDI*. 2019 (cited on pages 110, 121, 139, 140).

[262] Tao Xie, Nikolai Tillmann, Jonathan de Halleux, and Wolfram Schulte. « Fitness-Guided Path Exploration in Dynamic Symbolic Execution ». In: *DSN*. 2009 (cited on page 21).

[263] Babak Yadegari and Saumya Debray. « Symbolic Execution of Obfuscated Code ». In: *CCS*. 2015 (cited on page 20).

[264] Babak Yadegari, Brian Johannesmeyer, Ben Whitely, and Saumya Debray. « A Generic Approach to Automatic Deobfuscation of Executable Code ». In: *IEEE Symposium on Security and Privacy*. 2015 (cited on page 20).

[265] Mengjia Yan, Read Sprabery, Bhargava Gopireddy, Christopher W. Fletcher, Roy H. Campbell, and Josep Torrellas. « Attack Directories, Not Caches: Side Channel Attacks in a Non-Inclusive World ». In: *IEEE Symposium on Security and Privacy*. 2019 (cited on page 40).

[266] Hongseok Yang. « Relational Separation Logic ». In: *Theor. Comput. Sci.* 375.1-3 (2007) (cited on pages 7, 37).

[267] Junfeng Yang, Can Sar, Paul Twohey, Cristian Cadar, and Dawson R. Engler. « Automatically Generating Malicious Disks Using Symbolic Execution ». In: *IEEE Symposium on Security and Privacy*. 2006 (cited on page 20).

[268] Weikun Yang, Yakir Vizel, Pramod Subramanyan, Aarti Gupta, and Sharad Malik. « Lazy Self-Composition for Security Verification ». In: (2018) (cited on page 37).

[269] Zhaomo Yang, Brian Johannesmeyer, Anders Trier Olesen, Sorin Lerner, and Kirill Levchenko. « Dead Store Elimination (Still) Considered Harmful ». In: *USENIX Security Symposium*. 2017 (cited on pages 6, 9, 87–89, 104, 105, 107, 146).

[270] Jonas Zaddach, Luca Bruno, Aurélien Francillon, and Davide Balzarotti. « AVATAR: A Framework to Support Dynamic Security Analysis of Embedded Systems' Firmwares ». In: *NDSS*. 2014 (cited on page 20).

[271] Ziqiao Zhou, Michael K. Reiter, and Yinqian Zhang. « A Software Approach to Defeating Side Channels in Last-Level Caches ». In: *CCS*. 2016 (cited on page 54).

[272] Jean Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. « HACL*: A Verified Modern Cryptographic Library ». In: *CCS*. 2017 (cited on pages 4, 9, 41, 54, 56, 75, 76, 78, 79, 145).

# Appendix A

# Proofs

**Appendix overview**

This appendix details the proofs of the theorems and lemmas proposed in this thesis. The proofs of Chapter 4, concerning binary-level RelSE, are given in Section 4.4.3. The proofs of Chapter 6, concerning Haunted RelSE, are given in Section 6.3.5.

## A.1 Proofs of Chapter 4

**Section overview**

This section details the proofs of the theorems and lemmas of our relational symbolic execution for constant-time, proposed in Section 4.4.3:

- The proof of Lemma 1 is given in Appendix A.1.1,
- The proof of Lemma 2 is given in Appendix A.1.2
- The proof of Theorem 1, which states the correctness of our RelSE, is given in Appendix A.1.3,
- The proof of Theorem 4, which states that our RelSE is correct for bounded-verification of constant-time, is given in Appendix A.1.4.

### A.1.1 Proof of Lemma 1

Lemma 1 expresses that when the symbolic evaluation is stuck on a state $s_k$, there exist concrete configurations derived from $s_k$ which produce distinct leakages.

**Lemma 1.** *Let $s_k$ be a symbolic configuration obtained after $k$ steps. If $s_k$ is stuck, then there exists a model $M$ such that for each concrete configurations $c_k \approx_l^M s_k$ and $c'_k \approx_r^M s_k$, the execution from $c_k$ and $c'_k$ produce distinct leakages.*

Proof: Because $s_k$ is stuck, we know from Hypothesis 3 that an expression $\widehat{\varphi}$ is leaked and that $secLeak(\widehat{\varphi}, \pi)$ evaluates to false in the symbolic evaluation of $s_k$. We also know that there exists a model $M$ such that $M \vDash \pi \wedge \widehat{\varphi}_{|l} \neq \widehat{\varphi}_{|r}$. Let $c_k, c'_k$ be concrete configurations such that $c_k \approx_l^M s_k$, and $c'_k \approx_r^M s_k$. To show that the program is not constant-time at step $k$, that is $c_k \xrightarrow[t]{} c_{k+1}$ and $c'_k \xrightarrow[t']{} c'_{k+1}$ with $t \neq t'$, we proceed case by case on the symbolic evaluation, restricting to cases where $secLeak(\widehat{\varphi}, \pi)$ might evaluate to false. There are two main cases:

1. Symbolic execution is stuck on the evaluation of an expression,
2. Symbolic evaluation is stuck on the evaluation of an instruction.

**SE stuck on an expression.** First, we consider the case where the symbolic execution is stuck on the evaluation of an expression, restricting to the rule LOAD as other cases cannot be stuck.

— **Case** LOAD: In the symbolic execution, the expression `load` $e_{idx}$ is evaluated with $(\rho, \widehat{\mu}, \pi)\ e_{idx} \vdash \hat{\imath}$ and the index $\hat{\imath}$ is leaked. Assuming $\mathit{secLeak}(\hat{\imath}, \pi)$ evaluates to false with the model $M$, then $M(\hat{\imath}_{|l}) \neq M(\hat{\imath}_{|r})$. Moreover, because $c_k \approx_l^M s_k$ and $c'_k \approx_r^M s_k$, we have from Definition 10 that $c_k\ e_{idx} \vdash M(\hat{\imath}_{|l})$ and $c'_k\ e_{idx} \vdash M(\hat{\imath}_{|r})$. Because performing a step in the concrete execution leaks the value of $e_{idx}$, we have $c_k \xrightarrow[t \cdot M(\hat{\imath}_{|l})]{} c_{k+1}$ and $c'_k \xrightarrow[t' \cdot M(\hat{\imath}_{|r})]{} c'_{k+1}$ with $M(\hat{\imath}_{|l}) \neq M(\hat{\imath}_{|r})$, meaning that the execution is not constant-time at step $k$.

**SE stuck on an instruction.** Second, we consider the case where the symbolic evaluation is not stuck on the evaluation of an expression, but is stuck on the evaluation of an instruction. This can happen when evaluating rules STORE, ITE and I_JUMP.

— **Case** STORE: In the symbolic execution, the instruction `store` $e_{idx}\ e_{val}$ is evaluated with $(\rho, \widehat{\mu}, \pi)\ e_{idx} \vdash \hat{\imath}$ and the index $\hat{\imath}$ is leaked. Similarly as in case LOAD, we can show that we have $c_k \xrightarrow[t \cdot M(\hat{\imath}_{|l})]{} c_{k+1}$ and $c'_k \xrightarrow[t' \cdot M(\hat{\imath}_{|r})]{} c'_{k+1}$ with $M(\hat{\imath}_{|l}) \neq M(\hat{\imath}_{|r})$, meaning that the execution is not constant-time at step $k$.

— **Case** ITE-TRUE **and** ITE-FALSE: In the symbolic execution, the instruction $\mathit{ite}\ e\ ?\ l_{true} : l_{false}$ is evaluated with $(\rho, \widehat{\mu}, \pi)\ e \vdash \widehat{\varphi}$ and $\boldsymbol{eq_0}\ \widehat{\varphi}$ is leaked. Assuming $\mathit{secLeak}(\boldsymbol{eq_0}\ \widehat{\varphi}, \pi)$ evaluates to false with the model $M$, then $M(\widehat{\varphi}_{|l} = 0) \neq M(\widehat{\varphi}_{|r} = 0)$. Moreover, because $c_k \approx_l^M s_k$ and $c'_k \approx_r^M s_k$, we have from Definition 10 that $c_k\ e \vdash M(\widehat{\varphi}_{|l})$ and $c'_k\ e \vdash M(\widehat{\varphi}_{|r})$. Therefore in one of the concrete executions, $e$ evaluates to 0, the rule ITE-FALSE is applied and the location $l_{false}$ is leaked; whereas in the other execution, $e$ does not evaluate to 0, the rule ITE-TRUE is applied and the location $l_{true}$ is leaked. Finally, we have $c_k \xrightarrow[t \cdot l_{true}]{} c_{k+1}$ and $c'_k \xrightarrow[t' \cdot l_{false}]{} c'_{k+1}$ (or $c_k \xrightarrow[t \cdot l_{false}]{} c_{k+1}$ and $c'_k \xrightarrow[t' \cdot l_{true}]{} c'_{k+1}$), meaning that the execution is not constant-time at step $k$.

— **Case** I_JUMP: In the symbolic execution, the instruction `goto` $e$ is evaluated with $(\rho, \widehat{\mu}, \pi)\ e \vdash \widehat{\varphi}$. Let $M(\widehat{\varphi}_{|l}) = \mathtt{bv_l}$ and $M(\widehat{\varphi}_{|r}) = \mathtt{bv_r}$. Assume $\mathit{secLeak}(\widehat{\varphi}, \pi)$ evaluates to false with the model $M$, then $\mathtt{bv_l} \neq \mathtt{bv_r}$. Moreover, because $c_k \approx_l^M s_k$ and $c'_k \approx_r^M s_k$, we have from Definition 10 that $c_k\ e \vdash \mathtt{bv_l}$ and $c'_k\ e \vdash \mathtt{bv_r}$. In the concrete execution, $l_l \triangleq \mathit{to\_loc}(\mathtt{bv_l})$ and $l_r \triangleq \mathit{to\_loc}(\mathtt{bv_r})$ are leaked—note that $\mathit{to\_loc}$ is defined for $\mathtt{bv_l}$ and $\mathtt{bv_r}$ from Hypothesis 3. Because $\mathtt{bv_l} \neq \mathtt{bv_r}$ and $\mathit{to\_loc}$ is a one-to-one correspondence, we have $l_l \neq l_r$. Therefore, we have $c_k \xrightarrow[t \cdot l_l]{} c_{k+1}$ and $c'_k \xrightarrow[t' \cdot l_r]{} c'_{k+1}$ with $l_l \neq l_r$, meaning that the execution is not constant-time at step $k$.

□

### A.1.2 Proof of Lemma 2

Lemma 2 expresses that symbolic evaluation does not get stuck up to $k$, then for each pair of concrete executions following the same path up to $k$, there exists a corresponding symbolic execution.

**Lemma 2.** *Let $s_0$ be a symbolic initial configuration for a program $\mathbb{P}$ that does not get stuck up to $k$. For every concrete states $c_0$, $c_k$, $c'_0$, $c'_k$ and model $M$ such that $c_0 \approx_l^M s_0 \ \wedge \ c'_0 \approx_r^M s_0$, if $c_0 \xrightarrow[t]{}^k c_k$ and $c'_0 \xrightarrow[t']{}^k c'_k$ follow the same path, then there exists a symbolic configuration $s_k$ and a model $M'$ such that:*

$$s_0 \rightsquigarrow^k s_k \ \wedge \ c_k \approx_l^{M'} s_k \ \wedge \ c'_k \approx_r^{M'} s_k$$

Proof: *(Induction on the number of steps $k$).* Case $k = 0$ is trivial.

Let $c_{k-1}$ and $c'_{k-1}$ be concrete configurations and $s_{k-1}$ a symbolic configuration for which the inductive hypothesis holds. We need to show that Lemma 2 still holds at step $k$, meaning that for each concrete steps $c_{k-1} \xrightarrow[t]{} c_k$ and $c'_{k-1} \xrightarrow[t']{} c'_k$, there exists a symbolic configuration $s_k$ and a model $M'$ such that $c_k \approx_l^{M'} s_k \wedge c'_k \approx_r^{M'} s_k$ holds. This amounts to show that:

(i) the location in $s_k$ is the same as the location in $c_k$ and $c'_k$,

(ii) there exists a model $M_k$ such that $M_k \vDash \pi_k$

(iii) for all expressions $e$, either symbolic evaluation gets stuck, or $\left(\rho_k, \widehat{\mu}_k\right) e \vdash \widehat{\varphi}$ and $M_k(\widehat{\varphi}_{|p}) = \texttt{bv} \iff c'_k \ e \vdash \texttt{bv}$.

We can proceed case by case on the concrete evaluation of $c_{k-1}$ and $c'_{k-1}$. Note that from Definition 10, $c_{k-1}$ and $c'_{k-1}$ are at the same program location and therefore need to evaluate the same instruction.

**Case** STORE**.** Consider that an instruction `store` $e_{idx}\ e_{val}$ is evaluated at step $k-1$. Let $\hat{\imath}$ be the symbolic index and $\hat{\nu}$ be the symbolic value, meaning that $\left(\rho, \widehat{\mu}, \pi\right) e_{idx} \vdash \hat{\imath}$ and $\left(\rho, \widehat{\mu}, \pi\right) e_{val} \vdash \hat{\nu}$. Item (i) directly follows from the concrete and the symbolic evaluation rules that both just increment the location by 1. Next, we build the new model $M_k$ as:

$$M_k \triangleq M[\widehat{\mu}_k \mapsto \langle m_l \,|\, m_r \rangle] \qquad \begin{array}{l} \text{where } m_l \triangleq M(\widehat{\mu}_{|l})[M(\hat{\imath}_{|l}) \mapsto M(\widehat{\nu}_{|l})] \\ \text{and } m_r \triangleq M(\widehat{\mu}_{|r})[M(\hat{\imath}_{|r}) \mapsto M(\widehat{\nu}_{|r})] \end{array}$$

Intuitively, $M_k$ is equal to the old model $M$ in which we add the new symbolic memory $\widehat{\mu}_k$, mapping to the concrete value of the old memory $M(\widehat{\mu})$ where the index $M(\hat{\imath})$ maps to the value $M(\widehat{\nu})$. Notice that $M_k \vDash \pi_k$ because $M \vDash \pi$ and we only added new definitions (thus not changing satisfiability) from $M$ and $\pi$ to $M_k$ and $\pi_k$. Thus Item (ii) holds. Finally, we show Item (iii) by induction on the structure of expressions: for any expression $e$, if symbolic evaluation does not get stuck then $M_k(\widehat{\varphi}_{|l}) = \texttt{bv} \iff c_k \ e \vdash \texttt{bv}$ holds (case of the right projection is analogous). Note that only the memory is updated from step $k-1$ to step $k$, meaning that $c_k$, $s_k$, and $M_k$ only differ from $c_{k-1}$, $s_{k-1}$ and $M$ on expressions involving the memory. Thus, we only need to consider the rule LOAD, as the proof for other rules directly follows from $c_{k-1} \approx_p^M s_{k-1}$, Definition 10, and the definition of $M_k$.

Assume an expression `load` $e$ such that $s_k$ does not get stuck and let $\left(\rho_k, \widehat{\mu}_k\right) e \vdash \widehat{\imath'}$ and $\left(\rho_k, \widehat{\mu}_k\right)$ `load` $e \vdash \widehat{\nu'}$. We show that if Item (iii) holds for the expression $e$, then it holds for the expression `load` $e$. Formally, we must show that if $M_k(\widehat{\imath'}_{|l}) = \texttt{bv}_{\texttt{idx}} \iff c_k \ e \vdash \texttt{bv}_{\texttt{idx}}$ then $M_k(\widehat{\nu'}_{|l}) = \texttt{bv}_{\texttt{val}} \iff c_k$ `load` $e \vdash \texttt{bv}_{\texttt{val}}$.

First, we can rewrite $M_k(\widehat{\nu'}_{|l})$ as

$$\begin{aligned} M_k(\widehat{\nu'}_{|l}) &= M_k(\mathit{select}(\widehat{\mu}_{k|l}, \widehat{\iota'}_{|l})) \text{ by symbolic rule LOAD} \\ &= M(\widehat{\mu}_{|l})[M(\hat{\iota}_{|l}) \mapsto M(\widehat{\nu}_{|l})][M_k(\widehat{\iota'}_{|l})] \text{ by def. of } M_k \end{aligned}$$

From this point, there are two cases, either (a) the address of the load is the same as the address of the previous store, (b) the address of the load is different from the address of the previous store.

(a) The address of the load is the same as the address of the previous store: $M_k(\widehat{\iota'}_{|l}) = M(\hat{\iota}_{|l})$, therefore $M_k(\widehat{\nu'}_{|l}) = M(\widehat{\nu}_{|l})$. From the induction hypothesis, the concrete index of the load evaluates to $M_k(\widehat{\iota'}_{|l})$, that is $c_k\ e \vdash M_k(\widehat{\iota'}_{|l})$ which can be rewritten as $c_k\ e \vdash M(\hat{\iota}_{|l})$. From concrete rule STORE and $c_{k-1} \approx_l^M s_{k-1}$, we know that the concrete memory from $c_{k-1}$ to $c_k$ is updated at index $M(\hat{\iota}_{|l})$ to map to the value $M(\widehat{\nu}_{|l})$. Thus, we have $c_k$ `load` $e \vdash M(\widehat{\nu}_{|l})$ and by rewriting, $c_k$ `load` $e \vdash M_k(\widehat{\nu'}_{|l})$. Therefore we have shown that $M_k(\widehat{\nu'}_{|l}) = \texttt{bv}_{\texttt{val}} \iff c_k$ `load` $e \vdash \texttt{bv}_{\texttt{val}}$.

(b) The address of the load is different from the address of the previous store: $M_k(\widehat{\iota'}_{|l}) \neq M(\hat{\iota}_{|l})$, therefore $M_k(\widehat{\nu'}_{|l}) = M(\widehat{\mu}_{|l})[M_k(\widehat{\iota'}_{|l})]$. From the induction hypothesis, the concrete index of the load evaluates to $M_k(\widehat{\iota'}_{|l})$, that is $c_k\ e \vdash M_k(\widehat{\iota'}_{|l})$. From concrete rule STORE, we know that the concrete memory from $c_{k-1}$ to $c_k$ is only updated at address $M(\hat{\iota}_{|l})$ and untouched at address $M_k(\widehat{\iota'}_{|l})$. Plus, we know from $c_{k-1} \approx_l^M s_{k-1}$ that address $M_k(\widehat{\iota'}_{|l})$ maps to $M(\widehat{\mu}_{|l})[M_k(\widehat{\iota'}_{|l})]$ in $c_{k-1}$. Therefore, in configuration $c_k$, index $M_k(\widehat{\iota'}_{|l})$ maps to $M(\widehat{\mu}_{|l})[M_k(\widehat{\iota'}_{|l})]$ which, by rewriting, leads to $c_k$ `load` $e \vdash M_k(\widehat{\nu'}_{|l})$. Therefore we have shown that $M_k(\widehat{\nu'}_{|l}) = \texttt{bv}_{\texttt{val}} \iff c_k$ `load` $e \vdash \texttt{bv}_{\texttt{val}}$.

**Case** I_JUMP**.** Consider that an instruction `goto` $e$ is evaluated at step $k-1$. Notice that $e$ evaluates to the same value in $c_{k-1}$ and $c'_{k-1}$ because both executions follow the same path, and let `bv` be this concrete value. Concrete rule I_JUMP just sets the next location to $l_c = \mathit{to\_loc}(\texttt{bv})$. Symbolic evaluation of rule I_JUMP, evaluates $e$ to a symbolic value $\widehat{\varphi}$, computes a model $M' \vDash_{\text{SMT}} \pi \wedge \widehat{\varphi}_{|l} \wedge \widehat{\varphi}_{|r}$ and sets the next location to $l_s = \mathit{to\_loc}(M'(\widehat{\varphi}_{|l}))$. Note that from Hypothesis 3 and because symbolic execution does not get stuck, the rule can be non-deterministically applied with any model $M'$ satisfying the constraint.

Therefore, we can apply the rule with $M' = M$ which gives $l_s = \mathit{to\_loc}(M(\widehat{\varphi}_{|l}))$. Moreover, from the hypothesis $c_{k-1} \approx_p^M s_{k-1}$ and Definition 10, we have $M(\widehat{\varphi}_{|l}) = \texttt{bv}$ so $l_s = \mathit{to\_loc}(\texttt{bv})$. Therefore we have shown how to make a symbolic step such that $l_{s_k} = l_{c_k}$ and Item (i) holds. Finally, Items (ii) and (iii) directly follow from $c_{k-1} \approx_p^M s_{k-1}$ and Definition 10 because symbolic and concrete evaluation of expressions are not modified by rule I_JUMP.

**Other cases.** **Case** S_JUMP is trivial as only the location is updated to the same static value in both concrete and symbolic evaluation. **Case** ASSIGN is similar to case STORE (and simpler because it only requires to reason about the value and not the index). Finally **Cases** ITE_TRUE **and** ITE_FALSE are a bit similar to case I_JUMP

and rely on the fact that both concrete executions follow the same path, therefore a symbolic rule (either ITE_TRUE or ITE_FALE) can be applied to match the execution of both $c_k$ and $c'_k$.

**Conclusion.** We have shown that we can perform a step in symbolic execution from $s_{k-1}$ to a state $s_k$, and there exists a model $M'$ such that $c_k \approx_l^{M_k} s_k$ and $c'_k \approx_r^{M_k} s_k$. □

### A.1.3 Proof of Theorem 1

Theorem 1 claims the correctness of our symbolic execution, meaning that for each symbolic execution and model $M$ satisfying the path predicate, the concretization of the symbolic execution with $M$ corresponds to a valid concrete execution.

**Theorem 1** (Correctness of RelSE)**.** *For every symbolic configurations $s_0$, $s_k$ such that $s_0 \rightsquigarrow^k s_k$ and for every concrete configurations $c_0$, $c_k$ and model $M$, such that $c_0 \approx_p^M s_0$ and $c_k \approx_p^M s_k$, there exists a concrete execution $c_0 \to^k c_k$.*

PROOF: *(Induction on the number of steps k).* Case $k = 0$ is trivial.

Consider symbolic configurations $s_0$, $s_{k-1} \triangleq (l, \rho, \hat{\mu}, \pi)$ for which the induction hypothesis holds. That is, for each model $M$ and configurations $c_0$, $c_{k-1}$ such that $c_0 \approx_p^M s_0$ and $c_{k-1} \approx_p^M s_{k-1}$, we have $c_0 \to^{k-1} c_{k-1}$. Let $s_k \triangleq (l_k, \rho_k, \hat{\mu}_k, \pi_k)$ be the symbolic state such that $s_{k-1} \rightsquigarrow s_k$. We need to show that for each model $M_k$ and configurations $c_0$ and $c_k$ such that $c_0 \approx_p^{M_k} s_0$ and $c_k \approx_p^{M_k} s_k$, we have $c_0 \to^k c_k$.

**Build steps 0 to $k-1$.** First, we build the concrete execution from steps 0 to $k-1$. Because $M_k \vDash \pi_k$ (from Definition 10) and $\pi$ is a sub-formula of $\pi_k$ (from symbolic evaluation), we have $M_k \vDash \pi$, thus we can build $c_{k-1}$ such that $c_{k-1} \approx_p^{M_k} s_{k-1}$. Similarly we can build $c_0$ such that $c_0 \approx_p^{M_k} s_0$. Finally, from the induction hypothesis, we have $c_0 \to^{k-1} c_{k-1}$.

**Build step $k-1$ to $k$.** Second, we build the concrete execution from steps $k-1$ to $k$: we need to show that there is a step from $c_{k-1}$ to $c_k$ where $c_k \approx_p^{M_k} s_k$. Because concrete semantics never gets stuck (Hypothesis 4) there is a state $c'_k \triangleq (l', r', m')$ such that $c_{k-1} \to c'_k$ and because the semantics is deterministic (Proposition 2), this state is unique. Thus we need to show that $c'_k = c_k$, that is $c'_k \approx_p^{M_k} s_k$. Because $M_k \vDash \pi_k$ (from $c_k \approx_p^{M_k} s_k$ and Definition 10), this amounts to show that:

(i) the location in $c'_k$ is the same as the location in $s_k$,

(ii) for all expression $e$, either symbolic evaluation gets stuck, or $(\rho_k, \hat{\mu}_k)\ e \vdash \hat{\varphi}$ and $M_k(\hat{\varphi}_{|p}) = \texttt{bv} \iff c'_k\ e \vdash \texttt{bv}$.

We can proceed case by case on the concrete evaluation of $c_{k-1}$. Note that from Definition 10, $c_{k-1}$ and $s_{k-1}$ are at the same program location and therefore both evaluate the same instruction.

**Case** STORE**.** Consider that an instruction `store` $e_{idx}\ e_{val}$ is evaluated at step $k-1$. Let $\hat{\imath}$ be the symbolic index and $\hat{\nu}$ be the symbolic value, meaning that $(\rho, \hat{\mu}, \pi)\ e_{idx} \vdash \hat{\imath}$ and $(\rho, \hat{\mu}, \pi)\ e_{val} \vdash \hat{\nu}$. Item (i) directly follows from the concrete and the symbolic evaluation rules that both just increment the location by 1. We prove Item (ii) by induction on the structure of expressions: for any expression $e$, if symbolic evaluation does not get stuck then $M_k(\hat{\varphi}_{|p}) = \texttt{bv} \iff c'_k\ e \vdash \texttt{bv}$ holds. Note that only the

memory is updated from step $k-1$ to step $k$, meaning that $c'_k$ and $s_k$ only differ from $c_{k-1}$ and $s_{k-1}$ on expressions involving the memory. Thus, we only need to consider the rule LOAD, as the proof for other rules directly follows from $c_{k-1} \approx_p^{M_k} s_{k-1}$ and Definition 10.

Assume an expression `load` $e$ such that $s_k$ does not get stuck and let $\left(\rho_k, \widehat{\mu}_k\right) e \vdash \widehat{\iota'}$ and $\left(\rho_k, \widehat{\mu}_k\right)$ `load` $e \vdash \widehat{\nu'}$. We show that if Item (ii) holds for the expression $e$, then it holds for the expression `load` $e$. Formally, we must show that if $M_k(\widehat{\iota'}_{|p}) = \texttt{bv}_{\texttt{idx}} \iff c'_k\ e \vdash \texttt{bv}_{\texttt{idx}}$ then $M_k(\widehat{\nu'}_{|p}) = \texttt{bv}_{\texttt{val}} \iff c'_k$ `load` $e \vdash \texttt{bv}_{\texttt{val}}$.
First, we can rewrite $M_k(\widehat{\nu'}_{|p})$ as

$$\begin{aligned} M_k(\widehat{\nu'}_{|p}) &= M_k(\mathit{select}(\widehat{\mu}_{k|p}, \widehat{\iota'}_{|p})) \text{ by symbolic rule LOAD} \\ &= M_k(\mathit{select}(\mathit{store}(\widehat{\mu}_{k-1|p}, \hat{\iota}_{|p}, \widehat{\nu}_{|p}), \widehat{\iota'}_{|p})) \text{ by symbolic rule STORE} \end{aligned}$$

From this point, there are two cases, either (a) the address of the load is the same as the address of the previous store, (b) the address of the load is different from the address of the previous store.

(a) The address of the load is the same as the address of the previous store, i.e. $M_k(\widehat{\iota'}_{|p}) = M_k(\hat{\iota}_{|p})$. Therefore $M_k(\widehat{\nu'}_{|p}) = M_k(\widehat{\nu}_{|p})$. From the induction hypothesis, the concrete index of the load evaluates to $M_k(\widehat{\iota'}_{|p})$, that is $c'_k\ e \vdash M_k(\widehat{\iota'}_{|p})$ which can be rewritten as $c'_k\ e \vdash M_k(\hat{\iota}_{|p})$. From concrete rule STORE and $c_{k-1} \approx_p^{M_k} s_{k-1}$, we know that the concrete memory from $c_{k-1}$ to $c'_k$ is updated at index $M_k(\hat{\iota}_{|p})$ to map to the value $M_k(\widehat{\nu}_{|p})$. Thus, we have $c'_k$ `load` $e \vdash M_k(\widehat{\nu}_{|p})$ and by rewriting, $c'_k$ `load` $e \vdash M_k(\widehat{\nu'}_{|p})$. Therefore we have shown that:

$$M_k(\widehat{\nu'}_{|p}) = \texttt{bv}_{\texttt{val}} \iff c'_k \texttt{ load } e \vdash \texttt{bv}_{\texttt{val}}$$

(b) The address of the load is different from the address of the previous store, i.e. $M_k(\widehat{\iota'}_{|p}) \neq M_k(\hat{\iota}_{|p})$. Therefore:

$$M_k(\widehat{\nu'}_{|p}) = M_k(\mathit{select}(\widehat{\mu}_{k-1|p}, \widehat{\iota'}_{|p})) = M_k(\widehat{\mu}_{k-1|p})[M_k(\widehat{\iota'}_{|p})]$$

From the induction hypothesis, the concrete index of the load evaluates to $M_k(\widehat{\iota'}_{|p})$, that is $c'_k\ e \vdash M_k(\widehat{\iota'}_{|p})$. From concrete rule STORE, we know that the concrete memory from $c_{k-1}$ to $c'_k$ is only updated at address $M_k(\hat{\iota}_{|p})$ and untouched at address $M_k(\widehat{\iota'}_{|p})$. Plus, we know from $c_{k-1} \approx_p^{M_k} s_{k-1}$ that address $M_k(\widehat{\iota'}_{|p})$ maps to $M(\widehat{\mu}_{k-1|p})[M_k(\widehat{\iota'}_{|p})]$ in $c_{k-1}$. Therefore, in configuration $c'_k$, address $M_k(\widehat{\iota'}_{|p})$ also maps to $M(\widehat{\mu}_{k-1|p})[M_k(\widehat{\iota'}_{|p})]$ which, by rewriting, leads to $c'_k$ `load` $e \vdash M_k(\widehat{\nu'}_{|p})$. Therefore we have shown that:

$$M_k(\widehat{\nu'}_{|p}) = \texttt{bv}_{\texttt{val}} \iff c'_k \texttt{ load } e \vdash \texttt{bv}_{\texttt{val}}$$

**Case** I_JUMP**.** Consider that an instruction `goto` $e$ is evaluated at step $k-1$. Let $\widehat{\varphi}$ be the symbolic value of $e$, meaning that $\left(\rho, \widehat{\mu}, \pi\right) e \vdash \widehat{\varphi}$. Symbolic rule I_JUMP sets the location to $l_{s_k} = \mathit{to\_loc}(M(\widehat{\varphi}_{|l}))$, which is equal to $\mathit{to\_loc}(M(\widehat{\varphi}_{|r}))$ because $M$ satisfies the constraint $\widehat{\varphi}_{|l} = \widehat{\varphi}_{|r}$. Concrete rule I_JUMP evaluates expression $e$ to

a concrete value bv and sets the location to $l_{c_k} = to\_loc(\texttt{bv})$. From the hypothesis $c_{k-1} \approx_p^{M_k} s_{k-1}$ and Definition 10 we have $\texttt{bv} = M_k(\widehat{\varphi}_{|p})$, therefore $l_{c_k} = l_{s_k}$.

Finally, Item (ii) directly follows from $c_{k-1} \approx_p^{M_k} s_{k-1}$ and Definition 10 because symbolic and concrete evaluation of expressions are not modified by rule I_JUMP.

**Other cases.** **Case** S_JUMP is trivial as only the location is updated to the same static value in both concrete and symbolic evaluation. **Case** ASSIGN is similar to case STORE (and simpler because it only requires to reason about the value and not the index). Finally **Cases** ITE_TRUE **and** ITE_FALSE are similar to case I_JUMP.

**Conclusion.** We have shown that $c_{k-1} \to c'_k$ with $s_k \approx_p^{M_k} c'_k$. Because $\approx_p^{M_k}$ is a tight relation, we have $c'_k = c_k$ and thus $c_{k-1} \to c_k$. Therefore, for each model $M_k$ and configuration $c_0$ and $c_k$ such that $c_0 \approx_p^{M_k} s_0$ and $c_k \approx_p^{M_k} s_k$, we have $c_0 \to^k c_k$. □

### A.1.4 Proof of Theorem 4

Theorem 4 claims that if symbolic execution does not get stuck due to a satisfiable insecurity query, then the program is constant-time.

**Theorem 4** (Bounded-Verification for CT)**.** *Let $s_0$ be a symbolic initial configuration for a program $\mathbb{P}$. If the symbolic evaluation does not get stuck up to $k$, then $\mathbb{P}$ is constant-time up to $k$ w.r.t. $s_0$. Formally, if $s_0 \rightsquigarrow^k s_k$ then for all initial configurations $c_0$ and $c'_0$ and model $M$ such that $c_0 \approx_l^M s_0$, and $c'_0 \approx_r^M s_0$,*

$$c_0 \xrightarrow[t]{}^k c_k \ \wedge\ c'_0 \xrightarrow[t']{}^k c'_k \implies t = t'$$

*Additionally, if $s_0$ is fully symbolic and the execution does not get stuck for any $k$, then $\mathbb{P}$ is constant-time.*

PROOF: *(Induction on the number of steps $k$).* Case $k = 0$ is trivial.

Let $s_0$ be an initial symbolic configuration for which the symbolic evaluation never gets stuck. Let us consider a model $M_0$ and concrete configurations $c_0 \approx_l^{M_0} s_0$, $c'_0 \approx_r^{M_0} s_0$, for which the induction hypothesis holds at step $k-1$, meaning that for all $c_{k-1} \triangleq (l, r, m)$ and $c'_{k-1} \triangleq (l', r', m')$ such that $c_0 \xrightarrow[t]{}^{k-1} c_{k-1}$, $c'_0 \xrightarrow[t']{}^{k-1} c'_{k-1}$, then $t = t'$. We show that Theorem 4 still holds at step $k$.

From Lemma 2, there exists a model $M$ and a symbolic configuration $s_{k-1} \triangleq (l_s, \rho, \widehat{\mu}, \pi)$ such that:

$$s_0 \rightsquigarrow^{k-1} s_{k-1} \ \wedge\ c_{k-1} \approx_l^M s_{k-1} \ \wedge\ c'_{k-1} \approx_r^M s_{k-1} \tag{A.1}$$

We show by contradiction that the leakages bv and bv′ produced by $c_{k-1} \xrightarrow[\texttt{bv}]{} c_k$ and $c'_{k-1} \xrightarrow[\texttt{bv}']{} c'_k$ are equal. Note that from Eq. (A.1) and Definition 10, we have $l_s = l = l'$, therefore the same instruction and expressions are evaluated in configurations $c_{k-1}, c'_{k-1}$, and $s_{k-1}$. Suppose that $c_{k-1}$ and $c'_{k-1}$ produce distinct leakages. This can happen during the evaluation of rules LOAD, I_JUMP, ITE_TRUE, ITE_FALSE, STORE.

**Case** LOAD**.** Concrete evaluation of an expression `load` $e$ in configurations $c_{k-1}$ and $c'_{k-1}$ produces leakages bv and bv′ and, assuming the load is insecure, we have $\texttt{bv} \neq \texttt{bv}'$. Symbolic evaluation evaluates the index to a symbolic expression $\hat{\imath}$ and ensures $secLeak(\hat{\imath}, \pi)$ holds. From Eq. (A.1) and Definition 10, we have $M \vDash \pi$,

$\mathtt{bv} = M(\hat{\iota}_{|l})$ and $\mathtt{bv}' = M(\hat{\iota}_{|r})$. Because we assumed $\mathtt{bv} \neq \mathtt{bv}'$, then $M(\hat{\iota}_{|l}) \neq M(\hat{\iota}_{|r})$. Therefore, we have $M \vDash \pi \wedge \hat{\iota}_{|l} \neq \hat{\iota}_{|r}$, meaning that $secLeak(\hat{\iota}, \pi)$ evaluates to false and the symbolic execution is stuck, which is a contradiction. Therefore $\mathtt{bv} = \mathtt{bv}'$.

**Cases** I_JUMP, ITE_TRUE, ITE_FALSE, STORE. The reasoning is analogous.

**Conclusion.** We have shown that the hypothesis holds at step $k$. If $s_0 \rightsquigarrow^k s_k$, then for all model $M$ and initial configurations $c_0 \approx_l^M s_0$ and $c_0' \approx_r^M s_0$ such that $c_0 \xrightarrow[t]{}^{k-1} c_{k-1} \xrightarrow[t_k]{} c_k$ and $c_0' \xrightarrow[t']{}^{k-1} c_{k-1}' \xrightarrow[t_k']{} c_k'$ where $t = t'$, then $t \cdot t_k = t' \cdot t_k'$. □

## A.2 Proofs of Chapter 6

**Section overview**

This section details the proof of Theorem 6 presented in Section 6.3.5. The proof for Spectre-PHT is given in Appendix A.2.1 whereas the proof for Spectre-STL is given in Appendix A.2.2.

**Notations.** Let $C.f$ denote the field $f$ in the symbolic configuration $C$. For instance, for $C \triangleq \left(l, \widetilde{\delta}, \rho, \widehat{\mu}, \pi, \widetilde{\pi}, \widetilde{\lambda}\right)$, $C.\widetilde{\pi}$ is the speculative path predicate $\widetilde{\pi}$.

Let $C_{0\ldots n}$ denote a set of $n$ configuration $\{C_0 \ldots C_n\}$. Additionally, for a set of $n$ sets $\{S_0 \ldots S_n\}$, $S_{0\ldots n}$ denotes the union of these sets.

Let $\mathcal{M}(\widehat{\varphi_i}, \pi)$ be the set of values (i.e. relational concrete bitvectors) that $\widehat{\varphi_i}$ can take to satisfy $\pi$. More precisely,

$$\mathcal{M}(\widehat{\varphi_i}, \pi) = \{\widehat{\mathtt{bv}} \mid \vDash \pi \wedge \widehat{\mathtt{bv}} = \widehat{\varphi_i}\}$$

**Explicit RelSE.** In Explicit RelSE, there is no speculative path predicate $\widetilde{\pi}$, nor transient load set $\widetilde{\lambda}$, but just an *invalidation depth* $\gamma$ at which the transient paths are terminated (initially set to $+\infty$). Therefore, an explicit configuration is of the form $\left(l, \rho, \pi, \widehat{\mu}, \widetilde{\delta}, \gamma\right)$, where $l$, $\rho$, $\pi$, $\widehat{\mu}$, and $\widetilde{\delta}$ are defined as in Haunted RelSE (cf. Section 6.3).

Main changes between Haunte RelSE and Explicit RelSE happen in rules LOAD, ITE-TRUE, and ITE-FALSE and are given in Figure A.1. Additionally, we adapt the rule STORE defined in Figure 6.5, so that the index $\hat{\iota}$ does not leak in transient execution. Consequently, $secLeak(\hat{\iota}, E.\pi)$ is only evaluated in sequential execution, i.e. when $E.\gamma = +\infty$. Detailed explanations of the rules in Figure A.1 follows:

— LOAD non-deterministically chooses a value in the set of values that the load can take (defined by $lookup_{SB}$). It returns this value, together with the new invalidation depth—the minimum of the current invalidation depth $\gamma$ and the invalidation depth of the load $\gamma'$. Notice that this rule forks the symbolic execution for each possible value returned by $lookup_{SB}$. In contrast Haunted RelSE encodes this set of values in a single expression and does not fork the symbolic execution (cf. rule LOAD in Figure 6.4).

— ITE-TRUE is the evaluation of a conditional statement following the *true* branch. First, it evaluates the condition to a symbolic expression $\widehat{\varphi}$, and ensures that it can be leaked securely. Then, it computes two path predicates: $\pi_{true}$ in which $(true = \widehat{\varphi})$ and $\pi_{false}$ in which $(false = \widehat{\varphi})$. Finally, it non-deterministically chooses to follow the sequential path $((\pi', \gamma'') \triangleq (\pi_{true}, \infty))$ or to follow the

transient path ($(\pi', \gamma'') \triangleq (\pi_{false}, \delta)$). Notice that if both $\pi_{true}$ and $\pi_{false}$ are satisfiable, this rule forks the symbolic execution into two paths. Additionally, in this case, the symbolic execution forks into four paths because both ITE-TRUE and ITE-FALSE can be applied. In comparison Haunted RelSE only forks into two paths (one resulting from the application of ITE-TRUE and the other resulting from the application of ITE-FALSE). However if only one side of the condition is satisfiable then both Explicit and Haunted fork into two paths.

$$\text{LOAD}\ \frac{\left(l, \rho, \pi, \widehat{\mu}, \widetilde{\delta}, \gamma\right)\ e \vdash \hat{\imath} \qquad \boxed{(\widehat{\varphi}, \gamma') \in lookup_{SB}(SB(\widehat{\mu}, \widetilde{\delta}), Mem(\widehat{\mu}, \widetilde{\delta}), \hat{\imath})} \qquad secLeak(\hat{\imath}, \pi)}{\left(l, \rho, \pi, \widehat{\mu}, \widetilde{\delta}, \gamma\right)\ \texttt{load}\ e \vdash \widehat{\varphi}^{\widetilde{\delta}+\Delta}, \boxed{min(\gamma, \gamma')}}$$

$$\text{ITE-TRUE}\ \frac{\mathbb{P}[l] = \texttt{ite}\ e\ \texttt{?}\ l_{true}\ \texttt{:}\ l_{false} \qquad \left(l, \rho, \pi, \widehat{\mu}, \widetilde{\delta}, \gamma\right)\ e \vdash \widehat{\varphi}^{\delta}, \gamma' \qquad secLeak(\widehat{\varphi}, \pi) \qquad \boxed{\pi_{true} \triangleq \pi \wedge (true = \widehat{\varphi}_{|l} = \widehat{\varphi}_{|r})} \qquad \boxed{\pi_{false} \triangleq \pi \wedge (false = \widehat{\varphi}_{|l} = \widehat{\varphi}_{|r})} \qquad \boxed{(\pi, \gamma'') \in \{(\pi_{true}, \infty), (\pi_{false}, \delta)\}} \qquad \boxed{\vDash \pi'}}{\left(l, \rho, \pi, \widehat{\mu}, \widetilde{\delta}, \gamma\right) \rightsquigarrow_i \left(l_{true}, \rho, \pi', \widehat{\mu}, \widetilde{\delta} + 1, \boxed{min(\gamma, \gamma', \gamma'')}\right)}$$

$$\text{ITE-FALSE}\ \frac{\mathbb{P}[l] = \texttt{ite}\ e\ \texttt{?}\ l_{true}\ \texttt{:}\ l_{false} \qquad \left(l, \rho, \pi, \widehat{\mu}, \widetilde{\delta}, \gamma\right)\ e \vdash \widehat{\varphi}^{\delta}, \gamma' \qquad secLeak(\widehat{\varphi}, \pi) \qquad \boxed{\pi_{true} \triangleq \pi \wedge (true = \widehat{\varphi}_{|l} = \widehat{\varphi}_{|r})} \qquad \boxed{\pi_{false} \triangleq \pi \wedge (false = \widehat{\varphi}_{|l} = \widehat{\varphi}_{|r})} \qquad \boxed{(\pi', \gamma'') \in \{(\pi_{false}, \infty), (\pi_{true}, \delta)\}} \qquad \boxed{\vDash \pi'}}{\left(l, \rho, \pi, \widehat{\mu}, \widetilde{\delta}, \gamma\right) \rightsquigarrow_i \left(l_{false}, \rho, \pi', \widehat{\mu}, \widetilde{\delta} + 1, \boxed{min(\gamma, \gamma', \gamma'')}\right)}$$

FIGURE A.1 – Rules LOAD, ITE-TRUE and ITE-FALSE in Explicit RelSE. For readability, terms that are not important in the context are replaced with _ and differences with Haunted RelSE are highlighted with $\boxed{boxes}$.

**Definition 17** (Equivalence between Haunted and Explicit configurations ($\equiv$))**.** *Let $H$ and $E$ be symbolic configurations, respectively for Haunted RelSE and Explicit RelSE. These configurations are equivalent, denoted $H \equiv E$ if and only if:*

$$H.l = E.l \ \wedge\ H.\widetilde{\delta} = E.\widetilde{\delta} \ \wedge\ H.\rho = E.\rho \ \wedge\ H.\pi = E.\pi \ \wedge\ H.\widehat{\mu} = E.\widehat{\mu}$$

In Haunted RelSE the sets of possible load values are encoded in a single *ite* expression by the function $lookup_{ite}$ (cf. Algorithm 6), using unconstrained boolean variables (that the solver can freely set to *true* of $false$). The following property expresses that encoding the set of symbolic load values in a single *ite*-expression or considering each symbolic load value individually are equivalent. Moreover, setting the boolean variables used to build the *ite*-expression to $false$ restricts the set of expressions from which the load can take its value.

**Property 1** (Unconstrained *ite* are sets of expressions)**.** *Let $\pi$ be a path predicate and $\widehat{\varphi}$ be a symbolic if-then-else expression built over a set of relational symbolic bitvectors $\{\widehat{\varphi_1}, \ldots, \widehat{\varphi_n}\}$ and a set of booleans $\{\beta_1, \ldots, \beta_{n-1}\}$ such that:*

$$\widehat{\varphi} \triangleq (ite\ \beta_1\ \widehat{\varphi_1}\ (\ldots (ite\ \beta_{n-1}\ \widehat{\varphi_{n-1}}\ \widehat{\varphi_n})))$$

*Let* $\beta_{false} \subseteq \{\beta_1, \ldots, \beta_{n-1}\}$ *be the set of boolean variables that are set to* $false$ *in* $\pi$ *while others are left unconstrained; and let* $\widehat{\varphi}_{false} \subseteq \{\widehat{\varphi_1}, \ldots, \widehat{\varphi_{n-1}}\}$ *be the corresponding set of values in* $\widehat{\varphi}$*. Then* $\widehat{\varphi}$ *can take any symbolic value in the set* $\{\widehat{\varphi_1}, \ldots, \widehat{\varphi_n}\} \setminus \widehat{\varphi}_{false}$*.*

*Concretely* $\widehat{\varphi}$ *can take any concrete value in the set:*

$$\bigcup_{\widehat{\varphi_i} \in (\{\widehat{\varphi_1}, \ldots, \widehat{\varphi_n}\} \setminus \widehat{\varphi}_{false})} \mathcal{M}(\widehat{\varphi_i}, \pi)$$

**Theorem 6** (Equivalence Explicit and Haunted RelSE)**.** *Haunted RelSE detects a violation in a program if and only if Explicit RelSE detects a violation.*

Proof (Sketch). First, we show that Theorem 6 holds for Spectre-PHT in Appendix A.2.1 and second, that it holds for Spectre-STL Appendix A.2.2.

### A.2.1 Case Spectre-PHT

We show that Theorem 6 holds for detection of Spectre-PHT vulnerabilities, meaning that a violation is detected in Haunted RelSE if and only if it is detected in Explicit RelSE. We consider for this case that the rule LOAD only returns the sequential symbolic value in Explicit and Haunted RelSE. Consequently, the only rules that differ in Explicit RelSE and Haunted RelSE are the ITE-TRUE and ITE-FALSE rules.

**Proof overview.** In the following, we show that from two equivalent configurations $H$ and $E$, respectively for Haunted RelSE and Explicit RelSE, there is a violation in the execution from $H$ if and only if there is a violation in the execution from $E$. The proof goes by induction on the number steps in symbolic execution, and the goal is to show that states in Haunted RelSE and in Explicit RelSE following the same sequence of rules are equivalent. Notice that all the rules preserve the equivalence relation—except for the rules ITE-TRUE and ITE-FALSE. Moreover, as long as the equivalence relation is preserved, Theorem 6 holds. Therefore we only need to show that Theorem 6 still holds after the application of rules ITE-TRUE and ITE-FALSE.

Consider the application of the rule ITE-TRUE (case ITE-FALSE is analogous). The rule in Explicit RelSE produces two states $E_t$ and $E'_t$; whereas the rule in Haunted RelSE produces a single state $H_t$. Intuitively, the goal of the proof is to show that the path following $H_t$ is equivalent to the set of paths $\{E_t, E'_t\}$.

In the following, we focus on the **base case**, where initial states $E$ and $H$ correspond to sequential states, and show that after applying the rule ITE-TRUE once, the resulting state $H_t$ is equivalent to the pair of states $(E_t, E'_t)$. The general case considers the equivalence between a Haunted state $H'$ and a set of Explicit states $E_{1\ldots n}$. In particular, the general case requires to show that:

a) for all $1 \leq i \leq n$, $H'$ is equivalent to $E_i$ except on the value of the path predicate—which is important to derive Eq. (A.2),

b) the path predicate in $H$ is the disjunction of the path predicates in $E_{1\ldots n}$ ($H'.\pi = \bigvee_{0<i\leq n} E_i.\pi$)—which is important to prove the case LOAD,

c) when satisfiable, the sequential path predicate in $H'$ (i.e. $H'.\pi_{seq} \triangleq H'.\pi \wedge \bigwedge_{\widehat{\varphi} \in H'.\widetilde{\pi}} \widehat{\varphi}$) is equal to the path of the sequential state in $E_{1\ldots n}$ (i.e. the state $E_i \in E_{i\ldots n}$ such that $E_i.\gamma = \infty$) [1]—which is important to prove the case STORE.

1. It is also possible that the sequential path is infeasible, in which case $H'.\pi_{seq}$ is unsatisfiable and $E_i$ such that $E_i.\gamma = \infty$ does not exists.

**Evaluation of a conditional jump.** We consider here the base case where $E \equiv H$ and $E$ and $H$ are sequential states, meaning that $H.\widetilde{\pi} = \emptyset$, and $E.\gamma = \infty$. Consider that $E$ and $H$ are about to execute a conditional jump $\mathbb{P}[E.l] = \mathbb{P}[H.l] =$ `ite e ?` $l_{true}$`:` $l_{false}$. Because $H \equiv E$, `e` evaluates to the same symbolic value $\widehat{\varphi}^{\delta}$ in $H$ and $E$. Symbolic execution applies both rules ITE-TRUE and ITE-FALSE, proceeding as follows:

In Haunted RelSE (cf. Figure 6.5), the execution forks into two paths and gives the following states:

— A state $H_t$ following the *true* branch such that $H_t.\widetilde{\pi} := \{((true = \widehat{\varphi}), \delta) \cup H.\widetilde{\pi}\}$ and $H_t.l := l_{true}$ (by applying rule ITE-TRUE),

— A state $H_f$, following the $false$ branch such that $H_t.\widetilde{\pi} := \{((false = \widehat{\varphi}), \delta) \cup H.\widetilde{\pi}\}$ and $H_f.l := l_{false}$, (by applying rule ITE-FALSE).

In Explicit RelSE (cf. Figure A.1), the execution forks into four paths and gives the following states:

— Sequential *true* state $E_t$ such that $E_t.\pi := E.\pi \wedge (true = \widehat{\varphi})$ and $E_t.l := l_{true}$ (by applying rule ITE-TRUE with $\pi_{true}$),

— Sequential $false$ state $E_f$ such that $E_f.\pi := E.\pi \wedge (false = \widehat{\varphi})$ and $E_f.l := l_{false}$ (by applying rule ITE-FALSE with $\pi_{false}$),

— Transient *true* state $E'_t$ such that $E'_t.\pi := E.\pi \wedge (false = \widehat{\varphi})$, $E'_t.l := l_{true}$, and $E'_t.\gamma := \delta$ (by applying rule ITE-TRUE with $\pi_{false}$),

— Transient $false$ state $E'_f$ such that $E'_f.\pi := E.\pi \wedge (true = \widehat{\varphi})$, $E'_f.l := l_{false}$, and $E'_f.\gamma := \delta$, (by applying rule ITE-FALSE with $\pi_{true}$).

We can prove by induction on the number of steps in Haunted RelSE, that there is an equivalence between Haunted RelSE and Explicit RelSE configurations:

**EQ$_\mathbf{t}$** There is a vulnerability in execution following $H_t$ if and only if there is a vulnerability in execution following $E_t$ or in execution following $E'_t$.

**EQ$_\mathbf{f}$** There is a vulnerability in execution following $H_f$ if and only if there is a vulnerability in execution following $E_f$ or in execution following $E'_f$.

The proof for **EQ$_\mathbf{t}$** follows (case **EQ$_\mathbf{f}$** is analogous).

Because $H_t$, $E_t$ and $E'_t$ are equivalent except on the value of the path predicate $\pi$ they have the same current depth $\widetilde{\delta}$. First, we consider the case where $\widetilde{\delta}$ is below the retirement depth $\delta$ of the condition (Case $\widetilde{\delta} < \delta$). Then, we consider configurations such that the condition is retired (Case $\widetilde{\delta} \geq \delta$).

Note that because $H_t$, $E_t$ and $E'_t$ are equivalent except on the value of the path predicate $\pi$, the evaluation of expressions is the same in the three configurations because it only depends on $\rho$ and $\widehat{\mu}$. Therefore, for an expression $e$ we have:

$$H_t\ e \vdash \widehat{\psi} \iff E_t\ e \vdash \widehat{\psi} \iff E'_t\ e \vdash \widehat{\psi} \tag{A.2}$$

Additionally, they execute the same instruction. Therefore, we show that for rules LOAD, STORE, ITE-TRUE and ITE-FALSE, *secLeak* evaluates to $false$ in $H_t$ if and only if it evaluates to $false$ in $E_t$ or in $E'_t$.

**Case $\widetilde{\delta} < \delta$.**

— **Case** LOAD**.** Consider that $H_t$, $E_t$ and $E'_t$ all evaluate an expression `load` $e$. Let $\hat{\imath}$ be the symbolic value of the index, given by $H_t\ e \vdash \hat{\imath}$ in the first hypothesis

of rule LOAD (cf. Figure 6.4). From Eq. (A.2), we also have that $E_t\ e \vdash \hat{\imath}$ and $E'_t\ e \vdash \hat{\imath}$. We have to show that:

$$\neg secLeak(\widehat{\varphi}, H_t.\pi) \iff \neg secLeak(\widehat{\varphi}, E_t.\pi) \vee \neg secLeak(\widehat{\varphi}, E'_t.\pi)$$

From the definition of *secLeak* (cf. Section 4.4.1.1) we have the following:

- because $H_t.\pi = H.\pi$, we have $\neg secLeak(\hat{\imath}, H_t.\pi)$ if and only if $\vDash (H.\pi \wedge \hat{\imath}_{|l} \neq \hat{\imath}_{|r})$,
- because $E_t.\pi = E.\pi \wedge (true = \widehat{\varphi})$, we have $\neg secLeak(\hat{\imath}, E_t.\pi)$ if and only if $\vDash (E.\pi \wedge (true = \widehat{\varphi}) \wedge \hat{\imath}_{|l} \neq \hat{\imath}_{|r})$,
- because $E'_t.\pi = E.\pi \wedge (false = \widehat{\varphi})$, we have $\neg secLeak(\hat{\imath}, E'_t.\pi)$ if and only if $\vDash (E.\pi \wedge (false = \widehat{\varphi}) \wedge \hat{\imath}_{|l} \neq \hat{\imath}_{|r})$.

Therefore, $\neg secLeak(\hat{\imath}, E.\pi) \vee \neg secLeak(\hat{\imath}, E'.\pi)$ if an only if

$$\vDash \big(E.\pi \wedge ((true = \widehat{\varphi}) \wedge \hat{\imath}_{|l} \neq \hat{\imath}_{|r}) \vee ((false = \widehat{\varphi}) \wedge \hat{\imath}_{|l} \neq \hat{\imath}_{|r})\big)$$

which is equivalent to $\vDash (E.\pi \wedge \hat{\imath}_{|l} \neq \hat{\imath}_{|r})$.

From $E \equiv H$, we have $E.\pi = H.\pi$. Hence, there is a vulnerability in $H_t$ if an only if there is a vulnerability in $E_t$ or in $E'_t$.

— **Case** STORE**.** Consider that $H_t$, $E_t$ and $E'_t$ all evaluate an instruction `store` $e_{idx}\ e_{val}$. Let $\hat{\imath}$ be the symbolic value of the index, given by $H_t\ e_{idx} \vdash \hat{\imath}$ in the first hypothesis of rule STORE (cf. Figure 6.4). From Eq. (A.2), we also have that $E_t\ e \vdash \hat{\imath}$ and $E'_t\ e \vdash \hat{\imath}$.

Store indexes do not leak in transient execution, thus there is no check in $E'_t$. Therefore we have to show that:

$$\neg secLeak(\hat{\imath}, H_t.\pi_{ret}) \iff \neg secLeak(\hat{\imath}, E_t.\pi)$$

where $H_t.\pi_{ret}$ is the sequential path predicate in $H_t$, obtained by retiring the condition $\widehat{\varphi}$ from $H_t.\widetilde{\pi}$. Thus, $H_t.\pi_{ret} \triangleq H.\pi \wedge (true = \widehat{\varphi})$. From this, we have $\neg secLeak(\hat{\imath}, H_t.\pi_{ret})$ if an only if $\vDash (H.\pi \wedge (true = \widehat{\varphi}) \wedge \hat{\imath}_{|l} \neq \hat{\imath}_{|r})$.

For $E_t$ we have $\neg secLeak(\hat{\imath}, E_t.\pi)$ if an only if $\vDash (E.\pi \wedge (true = \widehat{\varphi}) \wedge \hat{\imath}_{|l} \neq \hat{\imath}_{|r})$.

From $E \equiv H$, we have $E.\pi = H.\pi$. Hence, we have shown that there is a vulnerability in $H_t$ if an only if there is a vulnerability in $E_t$.

— **Case** ITE-TRUE **and** ITE-FALSE**.** Evaluation of *secLeak* is similar as the evaluation of *secLeak* for load instructions.

**Case $\widetilde{\delta} \geq \delta$.** We now consider the case in which the invalidation depth $\delta$ of the condition $\widehat{\varphi}$ has been reached.

For Haunted RelSE, $H_t$ evaluates the rule STEP-RETIRE which calls the function *retire*ALL (cf. Figure 6.5 and Algorithm 9). The function *retire*ALL calls *retire*PHT (cf. Algorithm 3), which pops the condition $(true = \widehat{\varphi}, \delta)$ from $H_t.\widetilde{\pi}$, giving new symbolic state $H_{ret}$ such that $H_{ret}.\pi = H_t.\pi \wedge true = \widehat{\varphi}$.

For Explicit RelSE, the invalidation depth $E'_t.\gamma = \delta$ is reached and the path $E'_t$ is terminated, leaving only the sequential state $E_t$.

Note that, because $H \equiv E$, we have $H.\pi = E.\pi$, therefore $H_{ret}.\pi = E_t.\pi$. Moreover, because $H_t$ and $E_t$ are equivalent except on the value of the path predicate, we have $H_{ret} \equiv E_t$.

Finally, symbolic execution continues along equivalent states $H_{ret}$ and $E_t$, and because $H_{ret} \equiv E_t$, there is a vulnerability in states following $H_t$ if an only if there is a vulnerability in states following $E_t$.

### A.2.2 Case Spectre-STL.

We show that Theorem 6 holds for detection of Spectre-STL vulnerabilities, meaning that a violation is detected in Haunted RelSE if and only if it is detected in Explicit RelSE. We consider for this case that Explicit rules ITE-TRUE and ITE-FALSE (cf. Figure A.1) only return the sequential value and that ITE-TRUE and ITE-FALSE are replaced in Haunted RelSE by these modified Explicit rules. Consequently, the only rule that differ in Explicit RelSE and Haunted RelSE it the LOAD rule.

**Proof overview.** In the following, we show that from two equivalent configurations $H$ and $E$, respectively for Haunted RelSE and Explicit RelSE, there is a violation in the execution from $H$ if and only if there is a violation in the execution from $E$. The proof goes by induction on the number steps in symbolic execution, and the goal is to show that states in Haunted RelSE and in Explicit RelSE following the same sequence of rules are equivalent. Notice that all the rules preserve the equivalence relation—except for the rule LOAD. Moreover, as long as the equivalence relation is preserved, Theorem 6 holds. Therefore we only need to show that Theorem 6 still holds after the application of rules LOAD.

The evaluation of a LOAD rule, produces a set of $n$ states $E_{1...n}$ in Explicit RelSE; whereas in Haunted RelSE, it produces a single state $H'$. Intuitively, the goal of the proof is to show that the path following $H'$ is equivalent to the set of paths following $E_{1...n}$.

In the following, we focus on the **base case**, where initial states $E$ and $H$ correspond to sequential states, and show that after applying the rule LOAD once, the resulting state $H'$ is equivalent to the set of states $E_{1...n}$. The general case considers the equivalence between a Haunted state $H'$ and a set of Explicit states $E_{1...n}$ resulting from several applications of the LOAD rule. In particular, the general case requires to show that:

a) for all $1 \leq i \leq n$, $H'$ and $E_i$ are equivalent except on the values of the loads,

b) the set of symbolic values that a load can take in $H'$ corresponds to set of symbolic values that the load can take in $E_{1...n}$—which is important to derive Eq. (A.3),

c) the sequential value of the load in $H'$ (obtained by setting all the boolean variables in $H.\widetilde{\lambda}$ to false) is equal to the value of a the load in the sequential state in $E_{1...n}$ (i.e. the state $E_i \in E_{i...n}$ such that $E_i.\gamma = \infty$)—which is important to prove the case STORE.

**Evaluation of a load.** We consider here the base case where $E \equiv H$ and $E$ and $H$ are sequential states, meaning that $H.\widetilde{\pi} = \emptyset$, and $E.\gamma = \infty$. Consider that both $E$ and $H$ are about to evaluate a load expression `load` $e$ (cf. LOAD rules in Figure 6.4 and A.1). First, because $E \equiv H$, they both evaluate the index to the same symbolic value $\hat{\imath}$: $E\ e \vdash \hat{\imath}$ and $H\ e \vdash \hat{\imath}$.

In Explicit RelSE, the value of the load $(\widehat{\varphi}, \gamma')$ can be any value from the set $lookup_{SB}(SB(\widehat{\mu}, \widetilde{\delta}), Mem(\widehat{\mu}, \widetilde{\delta}), \hat{\imath})$; whereas in Haunted RelSE, this set of value is encoded in a single expression using $lookup_{ite}$ (cf. Algorithm 6). Consider that this function $lookup_{SB}$ returns a set of $n$ values $\{(\widehat{\varphi_1}, \delta_1), \ldots, (\widehat{\varphi_{n-1}}, \delta_{n-1}), (\widehat{\varphi_n}, \infty)\}$ such that $\widehat{\varphi_n}$ is the sequential value.

Explicit RelSE, forks the symbolic execution in $n$ distinct states $E_1, \ldots, E_n$, respectively returning value $\widehat{\varphi_1}, \ldots, \widehat{\varphi_n}$ for the load evaluation, and such that $\forall 1 \leq i \leq n.\ E_i.\gamma = \delta_i$. Let $E_{1...n}$ denote this set of configurations.

Haunted RelSE, returns a unique state $H'$ where the load evaluates to a symbolic expression $\widehat{\varphi}'$ such that:

$$\widehat{\varphi}' \triangleq (ite\ \beta_1\ \widehat{\varphi_1}\ (\ldots(ite\ \beta_{n-1}\ \widehat{\varphi_{n-1}}\ \widehat{\varphi_n})))$$

and the set of transient loads is updated as:

$$H'.\widetilde{\lambda} := \{(\beta_1, \delta_1), \ldots, (\beta_{n-1}, \delta_{n-1})\}$$

We can prove by induction on the number of steps in Haunted RelSE, that there is an equivalence between the configuration following $H'$ and the set of configurations following $E_{1\ldots n}$. In other words, there is a vulnerability in the execution following $H'$ if an only if there is a vulnerability in an execution following one of the states in $E_{1\ldots n}$.

**Equivalence between $H'$ and $E_{1\ldots n}$.** Consider that $H'$ and $E_{1\ldots n}$ follow the same path (we explain how to handle conditional statements that introduce new paths and new constraints later, in **Case** ITE). Under this hypothesis, all configurations $H'$ and $E_{1\ldots n}$ execute the same instructions and only differ on the value of the load. In particular, the evaluation of an expression that does not depend on the load is the same in configurations $H'$ and $E_{1\ldots n}$. Moreover, the set of concrete values of the load—obtained from the symbolic values $\widehat{\varphi}_1, \ldots \widehat{\varphi}_n$—are the same in $H'$ and $E_{1\ldots n}$, meaning that:

$$\forall 1 \leq i \leq n.\ \mathcal{M}(\widehat{\varphi_i}, H'.\pi) = \mathcal{M}(\widehat{\varphi_i}, E'_i.\pi) \tag{A.3}$$

Therefore, we show that for rules LOAD, STORE, ITE-TRUE and ITE-FALSE, *secLeak* evaluates to $false$ in $H'$ if and only if it evaluates to $false$ in one of $E_{1\ldots n}$. Let $\widehat{\psi}$ be the expression that is leaked in these rules. The case where $\widehat{\psi}$ does not depend on the load is trivial, as the value of $\widehat{\psi}$ is the same in $H'$, and configurations $E_{1\ldots n}$. Therefore, we only detail the case where $\widehat{\psi} = \widehat{\varphi}'$ in $H'$ and $\widehat{\psi} = \widehat{\varphi}_i$ in $E_{1\ldots n}$.

First, we consider configurations following $H'$ where none of the transient loads have been invalidated, meaning that all $\beta_i$ are unconstrained and all states following $E_i$ are still alive. Then, we consider configurations such that a transient load value has been invalidated (i.e. there is $\delta_i$ such that the current depth $\widetilde{\delta}$ is greater or equal than $\delta_i$).

**Case $\widetilde{\delta} < \delta_i$** We have to show for the rules LOAD, STORE, ITE-TRUE and ITE-FALSE that *secLeak* evaluates to $false$ in $H'$ if an only if it evaluates to $false$ in one of the states in $E_{1\ldots n}$.

— **Case** LOAD**.** Consider that $H'$, $E_{1\ldots n}$ all evaluate an expression `load` $e$ (cf. rules LOAD in Figure 6.4 and A.1). By hypothesis, $\widehat{\varphi}'$ is the symbolic value of the index in $H'$ (i.e. $H'\ e \vdash \widehat{\varphi}'$), and for all $1 \leq i \leq n$, $\widehat{\varphi}_i$ is the symbolic value of the index in $E_i \in E_{1\ldots n}$ (i.e. $E_i\ e \vdash \widehat{\varphi_i}$). We have to show that:

$$\neg secLeak(\widehat{\varphi}', H'.\pi) \iff \bigvee_{i=1}^{n} \neg secLeak(\widehat{\varphi}_i, E_i.\pi)$$

In Explicit RelSE, for $E_i \in E_{1\ldots n}$, we have $\neg secLeak(\widehat{\varphi}_i, E_i.\pi)$ if an only if $\vDash E_i.\pi \wedge \widehat{\varphi_{i|l}} \neq \widehat{\varphi_{i|r}}$. Let $\boldsymbol{BV}_i$ be the set of relational bitvectors values that $\widehat{\varphi_i}$ can take to satisfy $E_i.\pi$, that is $\boldsymbol{BV}_i = \mathcal{M}(\widehat{\varphi_i}, E_i.\pi)$. Consequently, we have $\bigvee_{i=1}^{n} \neg secLeak(\widehat{\varphi}_i, E_i.\pi)$ if an only if there is $\widehat{\mathtt{bv}} \in \boldsymbol{BV}_{1\ldots n}$ such that $\widehat{\mathtt{bv}_{|l}} \neq \widehat{\mathtt{bv}_{|r}}$.

Notice that because $lookup_{ite}$ declares a set of fresh unconstrained boolean variables (cf. Algorithm 6), and because we are in the case $\widetilde{\delta} < \delta_i$ (meaning that no value has been retired from $H'.\widetilde{\lambda}$ yet), all $\beta_i$ are unconstrained. Therefore, from Property 1, the symbolic index $\widehat{\varphi'}$ in Haunted RelSE can take any symbolic value in $\{\widehat{\varphi_1} \dots \widehat{\varphi_n}\}$. Consequently, $\widehat{\varphi'}$ can take any concrete value in the set:

$$\boldsymbol{BV'} = \bigcup_{i=1}^{n} \mathcal{M}(\widehat{\varphi_i}, H'.\pi)$$

and we have $\neg secLeak(\widehat{\varphi'}, H'.\pi)$ if an only if there is $\widehat{\mathtt{bv}} \in \boldsymbol{BV'}$ such that $\widehat{\mathtt{bv}}_{|l} \neq \widehat{\mathtt{bv}}_{|r}$.
From Eq. (A.3), we have $\boldsymbol{BV'} = \boldsymbol{BV}_{1\dots n}$, therefore, we have that *secLeak* evaluates to $false$ in $H'$ if an only if *secLeak* evaluates to $false$ in one of the states in $E_{1\dots n}$.

— **Case** STORE**:** Consider that $H'$, $E_{1\dots n}$ all evaluate an instruction `store` $e_{idx}$ $e_{val}$ (cf. rules STORE in Figure 6.4). By hypothesis, $\widehat{\varphi'}$ is the symbolic value of the index in $H'$ (i.e. $H'$ $e_{idx} \vdash \widehat{\varphi'}$) and for all $1 \leq i \leq n$, $\widehat{\varphi_i}$ is the symbolic value of the index in $E_i \in E_{1\dots n}$ (i.e. $E_i$ $e_{idx} \vdash \widehat{\varphi_i}$). Store instructions do not leak in transient execution so there is no insecurity check in $E_{1\dots n-1}$. Therefore we have to show that:

$$\neg secLeak(\widehat{\varphi'}, H.\pi) \iff \neg secLeak(\widehat{\varphi}_n, E_n.\pi)$$

In Explicit RelSE, for $E_n$, we have $\neg secLeak(\widehat{\varphi}_n, E_n.\pi)$ if an only if $\vDash E_n.\pi \wedge \widehat{\varphi_n}_{|l} \neq \widehat{\varphi_n}_{|r}$. Let $\boldsymbol{BV}_n = \mathcal{M}(\widehat{\varphi_n}, E_n.\pi)$ be the set of values that $\widehat{\varphi_n}$ can take to satisfy $E_n.\pi$. There is a vulnerability in $E_n$ if an only if there is $\widehat{\mathtt{bv}} \in \boldsymbol{BV}_n$ such that $\widehat{\mathtt{bv}}_{|l} \neq \widehat{\mathtt{bv}}_{|r}$.
In Haunted RelSE, for $H'$, the rule first computes the sequential path predicate $H'.\pi_{seq}$ by invalidating all transient loads in the transient load set $H'.\widetilde{\lambda}$, which gives $H'.\pi_{seq} \triangleq H.\pi \wedge \neg\beta_1 \wedge \dots \wedge \neg\beta_{n-1}$. Note that under this constraint, from Property 1, we have $\widehat{\varphi'} = \widehat{\varphi_n}$ and thus $\widehat{\varphi'}$ can take any value in $\boldsymbol{BV'} = \mathcal{M}(\widehat{\varphi'}, H'.\pi)$. There is a vulnerability in $H'$ if an only if there is $\widehat{\mathtt{bv}} \in \boldsymbol{BV'}$ such that $\widehat{\mathtt{bv}}_{|l} \neq \widehat{\mathtt{bv}}_{|r}$.
From Eq. (A.3), we have $\boldsymbol{BV'} = \boldsymbol{BV}_n$, therefore, we have that *secLeak* evaluates to $false$ in $H'$ if an only if *secLeak* evaluates to $false$ in $E_n$.

— **Case** ITE**:** Evaluation of *secLeak* is similar as the evaluation of *secLeak* for load instructions. For symbolic states resulting from ITE-TRUE and ITE-FALSE, we can show that there is an equivalence between state following the *true* branch; and states following the *else* branch. In other words, the state resulting from applying ITE-TRUE on $H'$ is equivalent to the set of states obtained by applying ITE-TRUE on $E_{1\dots n}$. Moreover, Eq. (A.3) still holds for these states because if adding the condition to the path predicate kills some states in $E_{1\dots n}$, it will also invalidate corresponding values in $H'$. The same holds for the rule ITE-FALSE.

**Case $\widetilde{\delta} \geq \delta_i$.** We now consider the case in which the invalidation depth of the transient value $\widehat{\varphi_i}$ has been reached.

For Haunted RelSE, $H'$ evaluates the rule STEP-RETIRE which calls the function *retire*ALL (cf. Figure 6.5 and Algorithm 9). The function *retire*ALL calls *retire*STL (cf. Algorithm 8), which invalidates the transient value $\widehat{\varphi_i}$ by removing the boolean $\beta_i$ from $H'.\widetilde{\lambda}$ and setting it to $false$ in the path predicate. This gives the new symbolic

state $H_{ret}$ such that $H_{ret}.\pi = H'.\pi \wedge \neg\beta_i$. From Property 1, this restricts the set of possible value for $\widehat{\varphi}'$ to $\{\widehat{\varphi}_0 \dots \widehat{\varphi}_n\} \setminus \{\widehat{\varphi}_i\}$.

For Explicit RelSE, $E_i.\gamma$ is reached and the path $E_i$ is terminated. This restricts the set of possible states to $E_{1\dots n} \setminus \{E_i\}$.

Finally symbolic execution continues along equivalent states $H_{ret}$ and $E_{1\dots n} \setminus \{E_i\}$, and in both cases the value of the load can now take any value in $\{\widehat{\varphi}_0 \dots \widehat{\varphi}_n\} \setminus \{\widehat{\varphi}_i\}$. □

# Appendix B

# Practical Challenges with BINSEC/REL and BINSEC/HAUNTED

**Appendix overview**

Implementing verification tools for constant-time and Spectre at binary-level poses a combination of challenges. This appendix presents some of the challenges we faced when implementing BINSEC/REL and BINSEC/HAUNTED, the solutions we adopted, and some opportunities for improvement:

- Standard challenges of binary-level analysis are discussed in Appendix B.1;
- Specification of the secret input, which is specific to information-flow; analysis at binary-level, is discussed in Appendix B.2;
- Difficulties to interpret the results are discussed in Appendix B.3;
- Validation of BINSEC/HAUNTED for detecting Spectre vulnerabilities, even though ground truth is not easily accessible, is discussed in Appendix B.4;
- Appendix B.5 concludes with general advice towards improving binary-level analyzers.

## B.1 Standard challenges in binary-level analysis

**Configuring initial memory.** In binary level symbolic execution, the initial memory is symbolic by default, but some parts must be initialized with information from the binary. For instance sections `.data` and `.rodata` contain unitialized data, and a load from one of these sections should read data directly from the binary. However, the case of the `.bss` section is trickier as it contains both variables initialized to 0—that we would like to set to 0—and uninitialized variables—that we would like to keep symbolic. Our solution is to keep the `.bss` section symbolic by default and, when necessary, to do some *reverse engineering* to specify the address ranges to initialize to 0.

**Deal with indirect functions.** Indirect functions [243] are functions whose implementation is chosen at runtime, using a resolver function. They are used in the GNU standard library (glibc), for instance, to implement multiple version of `memset` that are chosen at runtime depending on the CPU. In order to avoid analyzing multiple

implementations of indirect functions, we automatically replace calls to their resolver functions to calls to a specific implementation. For instance, for `memset`, we replace calls to the resolver function `__memset_ifunc` to calls to the specific implementation `__memset_ia32`.

**Limitations of BINSEC.** BINSEC symbolic execution engine does not have function summaries (a.k.a. stubs) for the standard library or system calls. Therefore, we only apply BINSEC/REL and BINSEC/HAUNTED to statically compiled binaries and stop a path when encountering a syscall [1]. BINSEC does not handle dynamically allocated memory, thus we have to statically allocate buffers.

## B.2 Specifying secrets

BINSEC/REL and BINSEC/HAUNTED have three different approaches for specifying secrets, offering different trade-offs between usability and realism.

### B.2.1 Reverse engineering

A first approach to specify secret input is to specify them as offsets from the initial stack pointer `esp`. This approach requires manual *reverse engineering* to identify secret and compute their offsets, as illustrated in Figure B.1.

```
out= dword ptr -20h
data= dword ptr -18h
key= dword ptr -10h

push    ebp
mov     ebp, esp
sub     esp, 20h
lea     eax, [ebp+key]
push    eax
lea     eax, [ebp+out]
push    eax
lea     eax, [ebp+data]
push    eax
call    encipher
```

FIGURE B.1 – Example of binary file disassembled with IDA. Let us call $esp_0$ the initial value of the stack pointer. Note that the `push` instruction increments `esp` by 4, thus `ebp` is set to $esp_0$+ 4. From there we can compute that symbolic secret input `key`, `data`, `out` are located at offsets `0x10 + 0x4`, `0x18 + 0x4`, `0x20 + 0x4` of $esp_0$.

**Pros and cons of the approach.**

*Pro:* Realistic and does not require any change in source code.

*Con:* Requires manual analysis and must be done each time the program is recompiled. Therefore it is not appropriate for large-scale experiments or for testing multiple compilers on the same program.

### B.2.2 Function stubs

A second approach to specify secrets directly in the source code is to use dummy functions as illustrated in Listing B.1. In symbolic execution, a call to `high_input_16(key)` is replaced by a function summary that initializes the memory at address `key` with 16 symbolic bytes considered as secret. This is the approach used in our experimental evaluation of BINSEC/REL (cf Section 4.6).

1. This only happens on our evaluation of BINSEC/HAUNTED, in the error-handling code of the stack protectors.

```
// Declare symbolic secret input
uint8_t key[16]; high_input_16(key);
uint8_t data[8]; high_input_8(data);
uint8_t out[8]; high_input_8(out);

// Function to analyze
encipher(data, out, key);
```

Listing B.1 – Specify secrets with dummy functions in C source code.

**Pros and cons of the approach.**

*Pro:* Does not require reverse-engineering and automatically applies to any binary compiled from the source, making it suitable for large-scale experiments or for testing multiple compilers.

*Con:* Requires to either modify the source code or to put a wrapper around the code to analyze (e.g. around the call to a library as illustrated in Listing B.1). Dummy function calls insert loads and stores which can introduce additional Spectre-STL violations, therefore this approach might not be ideal for studying Spectre-STL. For this reason, we do not use this approach in our experimental evaluation of BINSEC/HAUNTED.

### B.2.3 Global variables

A global variable in the source program is identified in the binary with a symbol that contains its name and address. In BINSEC/HAUNTED, a user can specify which global variables contain secret input, and BINSEC/HAUNTED will take care of initializing the corresponding addresses with symbolic secret values. This is the approach used in our experimental evaluation of BINSEC/HAUNTED.

**Pros and cons of the approach.**

*Pro:* Simple, automatic, and avoids introducing new STL-violations.

*Con:* Not very realistic as secret data would not be stored directly in the binary as global variables.

### B.2.4 Conclusion: specification at binary-level

Specifying security policies at binary level is more challenging as developers have to transpose their reasoning from source code to binary code (e.g. from program variables to memory addresses). Instrumentation at source level can improve automation and usability—which are necessary to run large scale experiments—but is not always realistic, or even possible. In BINSEC/REL and BINSEC/HAUNTED, we propose different approaches for specifying secrets that offer different trade-offs between usability and realism.

## B.3 Facilitate interpretation of counterexamples

Improving usability of Spectre-detection tools—in particular the interpretation of counterexamples—is crucial in order to facilitate their adoption, but also their

validation. This section, details the strategies implemented in BINSEC/REL and BINSEC/HAUNTED in order to facilitate the interpretation of counterexamples; and highlights potential opportunities for improvement.

**Counterexamples returned.** When detecting a violation, BINSEC/REL and BINSEC/HAUNTED return as a counterexample:

1. The instruction that leaks secrets and its location;
2. The initial configuration (memory and registers) that trigger the violation.

We also provide an IDA script to visualize the coverage of the analysis and highlight the violations found, allowing a user to directly identify the instructions triggering the violation in the assembly code.

**Spectre-STL.** For Spectre-STL, BINSEC/HAUNTED must additionally return the interleaving of loads and stores leading to the violation. Because the choice of loads and stores interleaving is encoded with boolean variables and left to the solver (as explained in Chapter 6), we have to encode this information in the formula and extract if from the model returned by the solver. To do this, we encode the address of loads and address of stores in the name of the boolean variables that encode the choices of the solver:

— For instance, if the boolean variable `load_08049d1c_from_08049cf5` is set to true by the solver, it means that the load at address `0x08049d1c` takes its value from the store at address `0x08049cf5`.

— Similarly, if the variable `load_08049d27_from_main-mem` is set to true, the load instruction at address `0x08049d27` takes its value from the initial memory.

**Further improving usability.** Improving usability is crucial for wide adoption of binary-level verification tools. We give possible leads to further improve the usability of BINSEC/REL and BINSEC/HAUNTED:

— In the memory configuration returned by our tools as a counterexample, users have to manually make the link between memory addresses and variables in the source code. Even though the reverse-engineering task is not difficult, it would enhance usability to automatically link memory locations to program variables when possible e.g. using symbols for global variables; or giving users the possibility to specify local variables of interest at the source level.

— BINSEC/HAUNTED could also easily improve the quality of counterexamples for Spectre-PHT by reporting information on the source of speculation—e.g. the address of the mispeculated conditionals, like in KLEESpectre.

— Finally, another improvement would be to differentiate whether a SCT violation occurs in sequential or in transient execution [2], as these two types of violations require different countermeasures.

## B.4 Validation of BINSEC/HAUNTED

Validating results from BINSEC/HAUNTED is challenging as there is no ground truth (especially for Spectre-STL), and SCT violations are difficult to find manually.

2. Recall that SCT [67] prevent leaks in both sequential and transient execution

**Litmus for Spectre-PHT.** To validate BINSEC/HAUNTED for Spectre-PHT, we mainly used the set of litmus tests developed by Paul Kocher [154] (precisely, the modified version from Pitchfork's benchmark [95] which is a set of 16 insecure simple test cases developed to test mitigations introduced by compilers. However, it still required manual analysis to precisely identify violations (e.g. number of vulnerabilities, locations, etc.). Additionally, we created a new set of secure litmus tests by applying the index-masking countermeasure to this initial set of litmus tests for Spectre-PHT.

**Litmus for Spectre-STL.** Validating BINSEC/HAUNTED for Spectre-STL was more challenging as there is no ground truth except for the initial proof-of-concept [3]. Moreover it is even more difficult to manually identify (or even confirm) vulnerabilities as it requires to reason about different load and store interleavings. Therefore, to validate BINSEC/HAUNTED, we manually crafted and documented 14 litmus tests for Spectre-STL [4].

**Cross validation.** For both Spectre-PHT and Spectre-STL, we compared BINSEC/HAUNTED against Pitchfork and KLEESpectre on these litmus test (when possible) and manually checked the results in case of deviation. Finally, we also use these litmus test for regression testing of BINSEC/HAUNTED.

**Connection to real Spectre attacks.** Finally, making the connection between real attacks and SCT violations discovered by the tools is an open question. Determining if SCT violations are exploitable requires a good understanding of the details of the micro-architecture and is micro-architecture-specific. A step to get closer to real attacks could be to extend SCT with details on the micro-architecture (e.g. adding conditions under which a processor may speculate), or include reasoning about attacker controlled input.

**Conclusion: validation of Spectre analyzers.** The lack of ground truth and the intricate nature of Spectre vulnerabilities (i.e. combination of speculation and side channels) makes it difficult to validate analyzers.

However, we believe that it would be difficult to provide a *precise and generic* benchmark for validation. Should such a benchmark be provided at source-level, the vulnerabilities in the binary-code would vary with compilation, and thus the benchmark would be imprecise; should it be provided at binary-level for a specific architecture, it would exclude LLVM-level tools, tools that do not handle the specific architecture, and tools that need to recompile from source to instrument the binary.

One way to help with validation is to improve the *usability* of the tools, in order to make interpretation of the results and cross validation easier.

## B.5 Conclusion

Binary-level reasoning introduces challenges that are not present in source-code analysis. For instance, the interpretation of the results is challenging as users have to transpose their reasoning from source code to binary code; or even cross-validation as other tools do not necessarily support the same architecture.

3. https://github.com/IAIK/transientfail/tree/master/pocs/spectre/STL
4. Open sourced at https://github.com/binsec/haunted_bench/blob/master/src/litmus-stl/programs/spectrev4.c

There is a tradeoff between realism of the analysis and usability. Most specification tasks that require reverse engineering—like configuring the initial memory or specifying secrets—can be partially automated with source-level instrumentation (e.g. using dummy functions), or using symbol information. This automation is crucial to be able to perform large scale experiments but is not always applicable to off-the-shelf binaries.

We believe that *improving usability and automation* is one of the keys to overcome these challenges. Indeed, usability and automation are crucial to run large scale experiments but also to be able to understand counterexamples returned by the analysis and can greatly help in prototype validation.

# Appendix C

# Details on Experimental Evaluation

**Appendix overview**

This appendix provides details on our experimental evaluations of BINSEC/REL and BINSEC/HAUNTED:

- It discusses the challenges of comparing BINSEC/HAUNTED with state-of-the-art tools, together with the solutions we adopted (cf. Appendix C.1);
- It details interesting use-cases (Appendix C.2);
- Finally, it provides some examples of our litmus tests for Spectre-STL (cf. Appendix C.2.3).

## C.1 Comparing BINSEC/HAUNTED against SoA: challenges and solutions

In Section 6.5.4, we compare BINSEC/HAUNTED against two state-of-the-art tools, Pitchfork [67] and KLEESpectre [252]. Our objective is not to compare tools per se, but to compare *underlying techniques*. Unfortunately the tools are very different and many details not related to the technique can impact performance, making the comparison challenging. This section details these challenges and our approaches to mitigate them (when applicable).

**LLVM vs. Binary.** While BINSEC/HAUNTED and Pitchfork operate at binary-level, KLEESpectre analyzes LLVM bitcode which gives it a performance advantage. This also means that the analyzed programs are different (`clang` LLVM vs. `gcc` binaries) and might contain different vulnerabilities (as shown in Section 4.6.2.3). Nevertheless, we tried to keep the analyzed files as close as possible by providing the same compilation options.

**System calls.** Symbolic analyzers might process system calls differently. For instance, `angr` (and thus Pitchfork) uses function summaries to model the effect of system calls on the symbolic state whereas BINSEC stops symbolic paths at syscalls. Because the performance of the tools eventually vary according to how they handle syscalls, we restrict our comparison to syscall-free programs `litmus-pht`, `litmus-pht-masked`, `litmus-stl`, `tea`, and `donna` and exclude `secretbox`, `ssl3-digest` and `mee-cbc`.

**Reported metrics.** While BINSEC/HAUNTED reports many information after its analysis (e.g. number of paths explored, number of instructions, number of queries sent to the solver, etc.), KLEESpectre and Pitchfork only report execution time and vulnerabilities found. Therefore the comparison restricts to these metrics and is less detailed than our controlled comparison of *Haunted RelSE* against *Explicit RelSE*, directly inside BINSEC/HAUNTED, given in Sections 6.5.2 and 6.5.3.

**Different properties.** The properties checked by KLEESpectre, Pitchfork and BINSEC/HAUNTED are not exactly the same.

First, KLEESpectre reports several types of gadgets but only one—leak secret (LS)—can actually leak secret data and is a violation of speculative constant-time, thus we only report LS gadgets found by KLEESpectre.

Second, KLEESpectre focuses on leakage from insecure loads and does not report leakage from secret-dependent branches (missing for instance vulnerabilities with AVX-based covert channels [225]), contrary to BINSEC/HAUNTED and Pitchfork. For this reason KLEESpectre fails to report one of the litmus test as insecure (i.e. `case_10`).

Third, KLEESpectre focuses on violations during transient execution while BINSEC/HAUNTED and Pitchfork also report violation during sequential execution. Because our benchmark is constant-time in sequential execution, this does not influence the number of violations found, however this may influence the execution time as Pitchfork and BINSEC/HAUNTED have more assertions to check.

Finally, Pitchfork reports secret-dependent store in transient execution as insecure contrary to BINSEC/HAUNTED and KLEESpectre which consider them secure as they are not committed to the cache [252].

**Different analysis techniques.** While KLEESpectre and BINSEC/HAUNTED are based on purely symbolic relational reasoning, Pitchfork is based on standard symbolic execution with *tainting*, which is faster but possibly incorrect.

**Different configurations.** For Spectre-STL, Pitchfork only supports reordering loads and stores in a window of 20 instructions, and does not allow to configure the size of the store buffer. In BINSEC/HAUNTED, we can configure the speculation window (set to 200) and the size of the store buffer (set to 20) [1]. While this is the closest configuration we can get, note that a load can bypass up to *20 stores* in BINSEC/HAUNTED, which makes the window larger than Pitchfork where loads can bypass up to *20 instructions*.

**Different implementation decisions.** First, we had to modify Pitchfork to enable verification of Spectre-STL without Spectre-PHT (which was not possible by default).

Second, KLEESpectre and BINSEC/HAUNTED report vulnerable instructions once, whereas Pitchfork may report many violations at a single instruction. Thus, we post-process the results of Pitchfork to report unique violations only. Note that it puts Pitchfork at a disadvantage because it still checks and reports these violations.

Third, Pitchfork stops a path after finding a violation, whereas BINSEC/HAUNTED continues the execution. To provide a closer comparison, we also consider a modified version of Pitchfork, namely Pitchfork-cont, which does not stop after finding a violation.

1. These are realistic values in modern processors.

Fourth, KLEESpectre fails to report an insecure litmus test (`case_7`) for no apparent reason. We suppose that they do not consider nested speculative executions but were only able to support this hypothesis by performing few small tests.

Finally, BINSEC/HAUNTED only considers indirect jump targets resulting from sequential execution and implements a shadow stack to constrain return instructions to their proper return site. Pitchfork does not implement this mechanism and follows transient indirect jump, leading to erratic behavior such as executing non-executable sections [2]. As a consequence, it reports 6 spurious violations in non executable `.data` section.

**Conclusion.** Comparing different tools is challenging as performance eventually depends on implementation details and might not reflect what we really want to measure—the underlying technique. Consequently such comparison must be taken with a pinch of salt.

For this reason, we believe that implementing our own *Explicit RelSE* baseline inside BINSEC/HAUNTED to compare against *Haunted RelSE* is a good solution, which makes it possible to compare very close implementations and focus on the underlying technique (cf. Sections 6.5.2 and 6.5.3).

In our experiments, we always tried our best not to put KLEESpectre and Pitchfork at a disadvantage (e.g. larger load reordering window in BINSEC/HAUNTED than in Pitchfork). However it was not always easy or possible—e.g. different performance due to different implementation decisions are hard to mitigate.

Finally, we did not encounter any difficulty to run Pitchfork and KLEESpectre on our own test cases, and adapting the implementation of Pitchfork for our comparison was quite simple, which is truly appreciable.

## C.2 Details on interesting use-cases

**Section overview**

This section details interesting use-cases that we analyzed in our experimental evaluation:

— Appendix C.2.1 details how to use BINSEC/REL to detect padding oracles, responsible for the famous Lucky13 [6] attack;

— Appendix C.2.2 illustrates the performance of BINSEC/HAUNTED on programs containing loops.

### C.2.1 Zoom on the Lucky13 Attack

Lucky 13 [6] is a famous attack exploiting timing variations in TLS CBC-mode decryption to build a Vaudenay padding oracle attack and enable plaintext recovery [6, 217]. Padding oracles happen for instance when timing variations depend on the length of the padding (i.e. padding must me checked in constant-time). We do not actually mount an attack but show how to find violations of constant-time that could potentially be exploited to mount such attack.

We focus on the function `tls-cbc-remove-padding` which checks and removes the padding of the decrypted record. We extract the vulnerable version from OpenSSL-1.0.1[6] (a excerpt is given in Listing C.1) and its patch from [10]. We set the length of the record data to 63 in both programs, meaning that the length of the padding can

2. This happened in two of our Spectre-STL litmus tests.

be found at `rec->data[62]`. Finally, we check with BINSEC/REL that no information is leaked during the padding check by specifying the record data as secret.

```
1 pad_len = rec->data[LEN-1]; // Get padding length
2 [...]
3 for (i = LEN - pad_len; i < LEN; i++)
4   if (rec->data[i] != pad_len)
5     return -1; // Incorrect padding
```

LISTING C.1 – Padding check in OpenSSL-1.0.1

**Results.** On the *insecure version*, BINSEC/REL accurately reports 5 violations, and for each violation, returns:

— the address of the faulty instruction,

— the execution trace which can be visualized with IDA,

— an input triggering the violation.

For instance, on the portion of code in Listing C.1, three violation are reported: two conditional statement depending on the padding length at line 3 and line 4, and a memory access depending on the padding length at line line 4. For the conditional at line line 3, BINSEC/REL returns in 0.11s the counterexample (`data_l[62]=0`, `data_r[62]=16`), meaning that an execution with a padding length set to 0 will take a different path that an execution with a padding length set to 16—i.e. the execution time is different when padding is set to 0, or set to 16.

On the *secure version*, BINSEC/REL explores all the paths in 438s and reports no vulnerability.

### C.2.2 Haunted RelSE for Spectre-PHT on programs with loops

We illustrate on litmus test `case_5` (Listing C.2) the role of Haunted RelSE for path pruning in programs containing loops. In this program the loop is bounded by the size of the array and can be fully unrolled in sequential execution. In transient execution, the loop can be mispeculated but unrolling is eventually bounded by the speculation depth. Performance of Explicit RelSE and Haunted RelSE are reported in Table C.1.

```
1 void case_5(uint64_t idx) {
2   int64_t i;
3   if (idx < publicarray_size)
4     for (i = idx - 1; i >= 0; i--)
5       temp &= publicarray2[publicarray[i] * 512];
6 }
```

LISTING C.2 – Litmus `case_5` where `publicarray_size` is set to 16.

RelSE restricted to in-order execution (NoSpec) produces 17 paths: a first path exits after the conditional at line 3, and 16 different path come from unrolling the loop 0 to 15 times.

Explicit RelSE forks into four paths after the conditional branch at line 3, two of them jumping on the loop at line 4. Then, each time the condition of the loop

| PHT | UInstr. | Paths | Time (s) | 🐞 |
|---|---|---|---|---|
| NoSpec | 305 | 17 | 1.3 | 0 |
| Explicit | 6824 | 407 | 26.5 | 1 |
| Haunted | 589 | 32 | 1.9 | 1 |

TABLE C.1 – Comparison of Explicit and Haunted RelSE for Spectre-PHT on litmus `case_5` where UInstr is the number of unrolled `x86` instructions.

is evaluated, Explicit RelSE forks again into four paths[3]. In total, 390 additional transient paths are explored (Table C.1).

The behavior of Haunted RelSE, is close to NoSpec: it only forks into two paths at line 3 and when the condition of the loop is evaluated. However, whereas NoSpec stops after 15 iterations of the loop, Haunted RelSE transiently executes the loop up to 15 times[4], which gives a total of 32 paths.

This example illustrates how Haunted RelSE can prune redundant paths compared to Explicit RelSE, achieving performance closer to standard (in-order) RelSE. *Haunted RelSE spares 375 paths compared to Explicit RelSE and is almost 14 times faster.*

### C.2.3 Litmus tests for Spectre-STL

In this section, we provide more details on our contribution regarding the litmus test suite for Spectre-STL. Paul Kocher proposed a set of litmus tests for Spectre-PHT [154], which verification tools could use in their test suite. However, no such set of litmus tests exists for Spectre-STL. We remedy this problem by proposing a similar set of 14 litmus tests for Spectre-STL, available at `https://github.com/binsec/haunted_bench/blob/master/src/litmus-stl/programs/spectrev4.c`. We provide an excerpt in Listing C.3 where cases 1, 2, 5, 8 are insecure and case 3 is secure.

3. Depending on the path predicate, either the four paths are satisfiable or only two of them are satisfiable.

4. The loop body is 14 instructions long and can therefore be speculatively executed 15 times in a speculation window of 200 instructions.

```
#define SIZE 16 // Size of arrays
uint8_t publicarray[SIZE] = { 1, ... ,16 }; // Public
uint8_t publicarray2[512 * 256] = { 20 };   // Public
uint8_t secretarray[SIZE];                  // Secret
/* Attacker's goal: learn any of the secret data in secretarray */

/* Based on original POC for Spectre-STL */
void case_1(uint32_t idx) { // Insecure
  register uint32_t ridx asm ("edx");
  ridx = idx & (secretarray_size - 1);
  uint8_t* data = secretarray;
  uint8_t** data_slowptr = &data;
  uint8_t*** data_slowslowptr = &data_slowptr;
  (*(*data_slowslowptr))[ridx] = 0; // Bypassed store
  temp &= publicarray2[data[ridx] * 512]; // Leak secret
}

/* Index masking is compiled to a store that can be bypassed */
void case_2(uint32_t idx) { // Insecure
  /* Compiled to a store & bypassed */
  idx = idx & (publicarray_size - 1);
  temp &= publicarray2[publicarray[idx] * 512]; // Leak secret
}

/* Index masking with index in a register */
void case_3(uint32_t idx) { // Secure
  register uint32_t ridx asm ("edx"); // Cannot be bypassed
  ridx = idx & (publicarray_size - 1);
  temp &= publicarray2[publicarray[ridx] * 512];
}

 /* Overwrite private pointer with public pointer */
uint8_t *ptr = secretarray;
void case_5(uint32_t idx) {
  register uint32_t ridx asm ("edx");
  ridx = idx & (array_size - 1); // Not bypassed
  ptr = publicarray;             // Bypassed store
  uint8_t toleak = ptr[ridx];
  temp &= publicarray2[toleak * 512]; // Leak secret
}

/* Overwrite index with multiplication to 0 */
uint32_t mult = 200;
void case_8(uint32_t idx) { // Insecure
  case8_mult = 0; // Bypassed store
  uint8_t toleak = publicarray[idx * mult];
  temp &= publicarray2[toleak * 512]; // Leak secret
}
```

Listing C.3 – Excerpt of our set of litmus tests for Spectre-STL where `case_1` is the original POC for Spectre-STL taken from https://github.com/IAIK/transientfail/blob/master/pocs/spectre/STL/main.c.