Optimizing Property-Preserving Compilation

Son Tuan Vu

SORBONNE UNIVERSITY

DOCTORAL THESIS

Optimizing Property-Preserving Compilation

Author:
Son Tuan VU

Supervisors:
Ms. Karine HEYDEMANN
Mr. Arnaud de GRANDMAISON
Mr. Albert COHEN

A thesis submitted in fulfillment of the requirements for the degree of Doctor of Philosophy

in the

Doctoral School of Computer Science, Telecommunication and Electronics (EDITE) of Paris
Computer Science Department

March 11, 2021

Abstract

Nowadays, software is of major importance to our societies: we use it in almost every aspect of our lives. The majority of software is written in high-level programming languages, then automatically translated and often optimized by a compilation chain into low-level machine code executable on the processor. However, target programs may present vulnerabilities, even when the source program was assumed or proven secure with regards to a given property (such as the presence of a security protection or secure coding patterns), primarily due to code optimizations performed during the compilation process. In order to ensure that binary compiled from a secure source program is still secure when it runs, engineers have to either try to preserve the security properties along the compilation flow, or verify these in the machine code using various binary analysis techniques, or both.

On the one hand, binary analysis performed on compiled programs usually requires properties about the program, for example ones describing the expected program behavior under a considered attack. These properties are naturally expressible as logical predicates referring to denotations defined in the source program; as a result, an instinctive way to provide binary analysis tools with these properties is to attach them to the program then request the compiler to preserve these down to the machine code. On the other hand, source-level security protections also need to be preserved by the compiler in order to produce binary as secure as the source program. In fact, these protections can be preserved if the compiler understands and maintains the underlying property associated to these. To this end, we need to instruct the compiler on the intention and preservation of these protections by specifying the associated properties. Therefore, property preservation is an important feature for compilers in order to ensure target program's security guarantees.

Unfortunately, property preservation is also a complicated feature to implement. Compiler optimizations—which focus solely on achieving better program performance, typically by reorganizing computation and removing "useless" code—are not suitable by design for preserving extra properties. The difficulty primarily lies in the fact that this information usually are not explicit in the high-level source language, thus compilers have no notion of the link between the extra properties and the code they refer to, and have no means to constrain transformations to preserve this link or to update the properties to adjust to any code transformation.

This thesis presents our proposed solutions to the problem of preserving properties introduced at the source level down to the machine code, throughout an optimizing compilation flow. The foundation of our approaches lies in the opacification technique, which hides information about an atom of operational semantics from compiler optimizations. We propose two different approaches with varying degrees of freedom for compiler optimizations, and provide a formal proof of correctness for the more lightweight (in terms of additional transformation constraints introduced) approach, as the latter may sound more sensitive to aggressive optimizations, so its effectiveness cannot directly be seen.

We implement both approaches in LLVM—an optimizing compilation framework widely-used in production—showing that preserving properties does not necessarily require significant modifications to compilers. We validate our approaches and their implementations on a range of security-sensitive benchmarks. The findings show that our approaches provide a novel, yet reliable means to preserve security-related properties throughout the optimizing compilation flow, while still allowing the compiler to perform aggressive optimizations. As a result, the work described in this thesis proposes a solution to the fundamental open issue in security engineering of preserving source-level countermeasures through optimizing compilation.

Contents

List of Figures

List of Tables

List of Abbreviations

ACSL	**A**NSI/ISO **C** **S**pecification **L**anguage
AES	**A**dvanced **E**ncryption **S**tandard
AST	**A**bstract **S**yntax **T**ree
CFG	**C**ontrol **F**low **G**raph
CFI	**C**ontrol **F**low **I**ntegrity
CPI	**C**ode **P**ointer **I**ntegrity
DAG	**D**irected **A**cyclic **G**raph
DIE	**D**ebugging **I**nformation **E**ntry
DWARF	**D**ebugging **W**ith **A**ttributed **R**ecord **F**ormats
Frama-C	**Fra**mework for **m**odular **a**nalysis of **C** programs
IoT	**I**nternet **o**f **T**hings
IFP	**I**nformation **F**low **P**reserving
IR	**I**ntermediate **R**epresentation
ISA	**I**nstruction **S**et **A**rchitecture
LTO	**L**ink-**T**ime **O**ptimization
MIR	**M**achine **I**ntermediate **R**epresentation
MLIR	**M**ulti-**L**evel **I**ntermediate **R**epresentation
RISC	**R**educed **I**nstruction **S**et **C**omputer
RSA	**R**ivest-**S**hamir-**A**dleman
SCI	**S**tep **C**ounter **I**ncrementation
SMT	**S**atisfiability **M**odulo **T**heories
SSA	**S**tatic **S**ingle **A**ssignment
WCET	**W**orst **C**ase **E**xecution **T**ime
WYSINWYX	**W**hat **Y**ou **S**ee **I**s **N**ot **Y**ou e**X**ecute

Chapter 1

Introduction

Software has become a crucial part of our everyday lives. It is ubiquitous: we use it in entertainment services, construction, transportation and warehousing, accommodation and food services, finance and insurance, health care and for so much more. The majority of software is written in high-level programming languages: they are easier and more efficient for programmers because they are closer to natural languages, so that programmers can tell more easily what the program will do. Programs written in human-readable high-level languages are then translated into low-level machine code, which finally gets executed on the processor.

The piece of software that does the translation between the program input by the programmer (called the source program) and the program to execute on the processor (called the target program) is called *compiler*. It is a crucial piece of software for programmers: only a correct compiler can ensure that the program will run according to what they have written. Moreover, over the course of its history, the duties of a compiler have significantly increased, from mere language translation to include program analysis, code optimization and so forth.

The concern with language translation is that programs written in high-level source languages implement some security guarantees, which unfortunately may not exist anymore in the translated programs. This problem usually comes from two different sources: (1) high-level programming languages offer security features to programmers in the form of type systems, module systems, encapsulation primitives and so forth, while most low-level machine languages run in commodity computers do not offer these same features; (2) programmers devise and implement security countermeasures to protect the programs against specific attacks, however these countermeasures do not affect the programs' behavior, and thus can be altered or removed by optimizations, because after all, the main responsibility of compilers remains translating a source program into a destination program that is equivalent in terms of behavior. In short, the problem is that what the programmer believes to be a secure program may become not secure once it is compiled. The question is then how can we ensure that what is a secure high-level program is still secure when it is compiled and it runs. To address this concern, a number of alternatives exist, for example software monitoring, software verification or secure compilation. This thesis is concerned with the two last ones. The first direction consists in verifying the security guarantees in compiled programs, which has lead to development of various analysis techniques to be performed on machine code; while the second direction aims at preserving the security guarantees along the compilation flow.

As will be explained in Section 1.1.1, software verification performed on compiled programs usually requires additional properties about the program such as the expected value of a loop counter at a loop exit or a range of possible values of an input variable. These properties can be naturally expressed at the source level by referring to denotations defined in the source program. However, they are not

meant to modify the program's semantics and thus should be introduced *externally* to the code itself, for example in the form of annotations or comments. Once the additional properties have been attached to the program, compilers need to convey these to the machine code.

Regarding secure compilation, as will be presented in Section 2.2.2, traditional approaches aim at producing destination programs that are as secure as their source counterparts, i.e. enjoys security properties such as confidentiality or integrity. However, we tackle the issue with a slightly different angle. We consider source programs hardened with some countermeasure schemes, and address the problem of producing destination programs containing these countermeasure schemes, which in turn enforce the security properties of confidentiality or integrity. To this end, we express implicit assumptions of a security countermeasure scheme as properties that refer to different denotations from the source program, then instruct compilers to preserve these down to machine code. These properties can usually be expressed in the form of assertion-like boolean expressions verifying the expected values of some variables at a specific program point. In contrast to the countermeasure schemes that are embedded into the program, these properties are not inherent in the code and should also be introduced externally to the program, in order to not interfere with the original program.

As a consequence, the problem considered in this thesis is that how can we reliably maintain and propagate source-level program properties, which are not part of the program's semantics, through the compilation process, down to machine code. As a matter of fact, preserving these properties throughout the compilation process is notoriously hard, notably in presence of optimizations. Optimizations have long been introduced into the compilation flow to produce more efficient machine code, and have since then improved the program's performance by leaps and bounds, to the point that they become compulsory in compilation. This is usually achieved by cleverly reorganizing computation and constantly removing code that they assume "unnecessary". On the downside, perhaps unsurprisingly, optimizations are by design not suitable for preserving additional program properties. This thesis focuses on the study of the interaction between compiler optimizations and the preservation of program properties, and presents our approach to reliably preserve properties throughout optimizing compilation flow. This chapter introduces the topic, explains the motivation and the challenges of this work, and presents the organization of the thesis.

1.1 Context and Motivation

In this section, we present the context and motivation of the work in this thesis from two different angles. We first approach the matter by introducing the needs for functional properties when carrying out analyses on executable binaries, then demonstrate the urge of preserving security properties through the compilation process, which has been a long-standing open issue in security engineering. This also explains the emphasis on security of the thesis.

1.1.1 Program Properties and Binary Analysis

Research in software engineering and computer security has led to new approaches for analyzing code for bugs and security vulnerabilities. There is a significant body of work devoted to testing, verifying and certifying the correctness of applications.

The focus of such work is to determine whether the program can reach a bad state. The common approach is to perform static analysis on *source code* written in a high-level language. However, this suffers from a major drawback, manifested practically as the so-called *What You See Is Not What You eXecute* (WYSINWYX) phenomenon (Balakrishnan and Reps, 2010): there is a mismatch between what a programmer intends (expressed in the source code) and what is actually executed by the processor (dictated by the binary code), notably due to compiler optimizations. As a consequence, although analyzes that are performed on source code are indeed a powerful approach to detect bugs and vulnerabilities, they fail to spot ones that are invisible in the source code and can only be detected by examining the binary code. In such a context, the importance of binary analysis is on the rise. In fact, in many situations, binary analysis is the only possible way to prove (or disprove) properties about the code that is *actually* executed. This explains the substantial amount of effort that the security research community has invested in developing analysis techniques to identify flaws in binary programs (Shoshitaishvili et al., 2016). Such analysis techniques vary widely in terms of the approaches used and the vulnerabilities targeted, but in general they face a common problem that might not be noticed at first glance: specification of the program's correct behavior in order to determine if it is robust with respect to the considered vulnerability. Propositional logic predicates—referring to denotations from the source program—provide a simple yet powerful way of expressing the soundness of a program and its behavior; we refer to these as *functional properties*.

For example, this is particularly true for robustness analysis against fault injection attacks. These attacks can alter the system's correct behavior by means of physical interference such as variations in the supply voltage, variations of the external clock, or lasers (Yuce, Schaumont, and Witteman, 2018; Bar-El et al., 2004). On micro-controllers and embedded processors, it has been observed, on different architectures and for different fault injection means, that such attacks can randomly corrupt a general-purpose register or skip the execution of an assembly instruction (Moro et al., 2013). In fact, the latter is a specific case of instruction replacement but the most frequent in practice (Moro et al., 2013), and is actually a major threat for embedded systems. As a countermove, various countermeasures have been proposed to protect programs against fault injections (Lalande, Heydemann, and Berthomé, 2014; Barry, Couroussé, and Robisson, 2016; Proy et al., 2017). These countermeasures then need to be verified and explained for certification purposes (Common Criteria, 2017). To this end, it is mandatory to have additional information about the property that must holds to ensure that the countermeasure is indeed effective and robust against the considered attack.

Let us consider a common smart-card application, shown in Listing 1.1. The `verifyPIN` function implements an authentication service through the comparison of an user-provided PIN code (`userPin`) against the fixed card PIN code (`cardPin`), by calling the utility function `byteArrayCompare` that returns 1 if PIN codes match and 0 otherwise (line 4). The user is allowed for a limited number of trials (`ptc`), initialized to 3 and reset upon authentication (line 5). The flag `auth` indicates whether the user has been authenticated and is initialized to 0.

A fault attack is considered successful if, for instance, the user is authenticated with an erroneous PIN. Assuming a 4-digit PIN code, the property describing the correct behavior of the application can be expressed using the following logic predicate `(auth == 0)|| (auth == 1 && userPin[0] == cardPin[0] && ... && userPin[3] == cardPin[3])` that needs to be verified right before the function returns, at the program point denoted by the label `verify_point` (line 11). More precisely, when the

```
1  unsigned verifyPIN()
2  {
3    if (ptc > 0) {
4      if (byteArrayCompare(userPin, cardPin) == 1) {
5        ptc = 3;
6        auth = 1;
7      } else {
8        ptc--;
9      }
10   }
11   verify_point: /* Property checkpoint */ ;
12   return auth;
13 }
```

LISTING 1.1: PIN authentication.

program is subjected to fault injections, if the predicate is evaluated at this program point and yields true, one can conclude that the program is robust with respect to the considered property.

It has been shown that the final assembly code placement and memory layout are of high importance, as some fault attacks may take advantage of them (Kelly, Mayes, and Walker, 2017). Therefore, to take these machine-level details into account, fault attack vulnerability assessment must be performed on the binary code (Bréjon et al., 2019; Given-Wilson et al., 2017; Laurent et al., 2019). As a result, the information about the considered property needs to be conveyed to the binary analysis evaluating the correctness or robustness of the program. Currently, these properties are either inserted manually at binary level (Goubet et al., 2015; Bréjon et al., 2019; Laurent et al., 2019)—which is tedious and error-prone—or communicated via code instrumentation such as `assert` (Given-Wilson et al., 2017). On the one hand, it is handy to express the program property at the source level, where programmers still have access to the high-level details of the program. On the other hand, relying on code instrumentation evaluating functional properties at run-time is only acceptable for evaluation and test and clearly not suitable for code used in production. Furthermore, compilers can make use of the code instrumentation to further optimize the program, which is not at all the intent of the programmer when writing the original program. In other words, it would be most beneficial to have an automatic way to provide binary analysis with direct access to the source-level properties, and this in a manner such that the properties do not interfere with the program itself but rather are external to the code. With that being said, the natural direction consists in propagating the functional properties, expressed at the source level, through the compilation pipeline down to machine code, in parallel with program transformations.

1.1.2 Program Properties and Security Countermeasures

Not only the WYSINWYX phenomenon justifies the needs for binary analysis, it also explains the so-called *correctness-security gap* of optimizing compilers, which arises when a compiler optimization preserves the functionality of but violates a security guarantees made by source code (D'Silva, Payer, and Song, 2015).

Let us illustrate this correctness-security gap through a well-known cryptography scenario. The leakage of sensitive data such as secret keys is a major threat, hence, programmers need to ensure the security property assuming the confidentiality of such a value in their applications. A common way to do this consists in erasing sensitive data from memory once they are no longer needed (Percival, 2014),

including keys, seeds of random generators, and temporary encryption or decryption buffers.

However, this may not be as easy as it seems: security engineers have been painfully fighting optimizing compilers to achieve their goal (Percival, 2014; Simon, Chisnall, and Anderson, 2018). Consider the example in Listing 1.2. The `secret` buffer containing sensitive information is allocated on the stack and should be erased before returning from the function; this security protection is implemented through a call to `memset()`. However, compilers will spot that `secret` goes out of scope, meaning that access after the function returns is an error or has unspecified behavior, hence will consider the call to `memset()` as unnecessary, removing the erasure as part of "dead store elimination". The source code is designed to be secure against exploits that access values that persist in memory, while the compiled code does not preserve this guarantee despite "dead store elimination" being a sound optimization and despite the code in `process_sensitive` having well-defined semantics.

```
1 void process_sensitive(void) {
2   uint8_t secret[32];
3   ...
4   memset(secret, 0, sizeof(secret));
5   verify_point: /* Property checkpoint */ ;
6 }
```

LISTING 1.2: Erasing a secret buffer on the stack.

At this point, we would like to emphasize that the security protection implies a property—expressed as the logic predicate `secret[0] == 0 && ... && secret[31] == 0`—that needs to be verified right before the function returns, at the program point denoted by the label `verify_point` (line 5). This thus hints at a more general challenge of preserving specific values at specific points of the program execution. These points and values are associated with protection schemes and countermeasures dictated by security concerns, and they have to be preserved for security reason, while also be traced down to machine code for verification purposes.

As will be shown in Section 3.3, applications are commonly secured by inserting protections at source level, but compilers may fail to implement the programmers' intentions as these protections often do not alter the program observable behavior, resulting in unsafe machine code. Interestingly, these protections usually involve implicit associated properties; we refer to these as *protection-derived properties*. If compilers manage to maintain these properties, then the corresponding security protections will also be preserved. With that being said, a natural approach to guarantee the presence of a given countermeasure scheme in the executable binary consists in specifying the protection-derived properties to the compiler and instructing the latter to propagate these properties down to machine code.

1.2 Challenges

Designing and implementing a property-preserving compilation framework is easier said than done. In fact, for additional properties which usually take the form of functional properties kept external to the code, besides propagating them through the compilation flow, we also have to take care of their consistency with the code undergoing transformations and optimizations, as well as their embedding into the compiled binary without interfering with the executable code itself. Compilers traditionally care about functional correctness and I/O effects: they have no notion of

the link between the extra properties and the code they refer to; they thus have no means to constrain transformations to preserve this link or to update the properties to adjust to any code transformation. Optimizing compilers tend to remove everything that is not behaviorally observable, and extra properties are obviously not supposed to modify the program logic. Hence compilers do not know how to maintain such information through the compilation flow. To further complicate matters, variables referenced in program properties may also be affected by compiler optimizations: an unused variable may be optimized out, i.e. completely removed from the transformed program, thus invalidating the semantics of a given property.

As for security properties, they are not naturally captured as logical expressions of the source program variables in general. However, it is possible to preserve them at the binary level by maintaining the source-level security countermeasures designed to enforce these properties. Unfortunately, as will be detailed in Section 2.2.3, Section 2.2.4 and Section 3.3, optimizing compilers are known to be unreliable on this aspect. We argue that a plausible solution consists in (1) devising a careful encoding of the security protection-derived properties as functions of source-level denotations or state (i.e. functional properties), as shown in the example from Section 1.1.2, then (2) leveraging the same mechanism of functional property preservation to preserve these down to machine code, thus preserving their associated security countermeasures.

Clearly, one of the biggest, if not the biggest challenge of propagating and preserving properties lies in the way we manage the interaction of properties with code optimizations. Perhaps the most straightforward approach (conception-wise but not implementation-wise) is to define new transformation rules for each optimization, taken into account the properties defined in source program. However, this is not at all suitable for implementing in realistic optimizing compilers, which generally include hundreds of different optimizations; tuning each and every one of these optimizations would be infeasible. As a result, we want to take a different route, seeking to design a generic, optimization-agnostic mechanism to preserve properties during compilation, so that it can be implemented in modern optimizing compilers.

1.3 Thesis Contributions

As explained in Section 1.1, there is evidently a need for a compilation framework that, given program properties expressed in the source level, propagates the information through the whole compilation process until the executable binary. The goal of such a framework is twofold: it provides (1) additional information—usually taking the form of functional properties external to the program itself—required by various analyses performed on the executable binary, and (2) a way to reliably specify security protection-derived properties and constrain compiler optimizations to preserve them, thus allowing effective implementation of the countermeasures.

The high-level contribution of this work is a **property- and optimization-agnostic approach for preserving program properties throughout the optimizing compilation flow**. Furthermore, we seek to build a property-preserving compilation framework by integrating our approach into an open-source, widely-used production compiler, for the use of a broader community including security and compilation. More specifically, the contributions of this thesis are:

- The construction of a first mechanism to propagate functional properties down to machine code, while still generating optimized code. To this end, we formally define the notion of functional property, and its preservation through

program transformations. We further show that the problem of functional property preservation can be reduced to preserving observations, and apply this latter to security countermeasure preservation.

- The design of a more lightweight mechanism to preserve observations throughout the compilation pipeline, *with minimal interference* with compiler optimizations. Similarly, we formally define the notion of observation and its preservation through program transformations. Moreover, we also provide a formal proof of the mechanism's correctness using a simplified intermediate program representation.
- The implementation of both mechanisms in a widely-used production compiler with no modification to individual optimization passes.

In fact, our observation preservation mechanism has a broad range of potential applications, some of which we explore in detail, others we only discuss. More specifically, we focus on leveraging the mechanism to reliably (1) propagate properties needed for testing, inspecting or verifying machine code and (2) maintain security protection-derived properties as a means to preserve the latter in machine code. The same mechanism can also be used to improve the debugging process, as will be discussed in Section 6.2.

1.4 Thesis Organization

Chapter 2 presents the existing work on preserving program properties during compilation. This ranges from preserving functional properties for improving optimizations (Section 2.1.1) or program analysis at binary level (Section 2.1.2 and Section 2.1.3), to secure compilation (Section 2.2) that seeks to preserve security properties down to machine code. Chapter 3 describes the background information required to support this work, as well as the different security use-cases that illustrate the open problem in security engineering that we try to solve in this thesis. Following that, Chapter 4 (resp. Chapter 5) describes our mechanisms to propagate and preserve functional properties (resp. observations) throughout an optimizing compilation flow, down to machine code. We also detail the implementation of these mechanisms in an existing compiler widely-used in academic and industrial settings, and validate our work on the motivating security use-cases presented in Chapter 3. Finally, Chapter 6 concludes the thesis and discusses different perspectives of this work.

Chapter 2

Related Work

This chapter presents a non-exhaustive state-of-the-art research work proposing different approaches to preserve properties throughout compilation. This also highlights the features required for a property-preserving compilation framework that we seek to design.

The first section presents preservation approaches dealing with functional properties, while the second section focuses on security properties. Finally, we conclude by describing some choices we have made which further clarify the scope of this work.

2.1 Preserving Functional Properties

Compilers have long used functional properties, usually in the form of hints and annotations, for various purposes. Generally speaking, this can be classified into three different categories, based on the goals of these uses by compilers: performance, safety analysis and security analysis.

2.1.1 Functional Properties for Performance Optimizations

An optimizing compiler is commonly structured as a sequence of passes. Each pass has a source program, which is analyzed and transformed to a target program, which then becomes the source for the next pass in the sequence. By augmenting the analysis phase of an optimization pass with information from additional program invariants, expressed as functional properties, it is possible to significantly enhance the quality and the effectiveness of optimization passes. These invariants may be supplied externally via a correctness proof or a static analysis of the source program. For example, consider a program which uses McCarthy's 91 function (Manna and McCarthy, 1969), which we write as `M91(x)`. The original function is doubly recursive, but has the simple property that, given an input `x`, the result is 91 if $x \leq 100$, and $(x - 10)$ otherwise. Suppose that this invariant is supplied from a correctness proof, external from the compiler analysis phase, a compiler may then replace an invocation of this function `y = M91(x);` with the following substantially simpler conditional statement: `y = (x <= 100)? 91 : (x - 10);`.

As previously stated in Section 1.2, the key technical challenge is to accurately propagate the properties through multiple optimization passes. The difficulty arises because an optimization may alter program structure in arbitrary ways. For instance, it can removes portions of the program, add fresh variables and statements, or reorder statement executions. Therefore, a property, in order to be usable by subsequent optimization passes, cannot simply be copied over unchanged from the source to the target program of an optimization.

An approach has been proposed to (initially) show that a program transformation is correct by augmenting a transformation with an auxiliary *witness generation* procedure (Namjoshi and Zuck, 2013). For every application of the transformation, the witness generator constructs a witness relation, between the source and the target programs of the considered transformation, which guarantees the correctness of that transformation instance. A witness relation connects the values of source and target variables at corresponding program locations. Typically, it encodes invariants about the source and target programs, which are enferred during the analysis phase of an optimization. For instance, "constant propagation" generates assertions about which variables of source program are constant at each program point, while "dead code elimination" depends on a liveness analysis that generates assertions about the live variables, also at each program point.

In passing, the authors leverage the witness relation to implement the propagation of externally supplied functional properties. They show that a source program invariant (which may be supplied externally or computed internally as part of the analysis phase of an optimization pass) can be propagated to the target program by computing its pre-image with respect to the witness relation. An implementation of the methodology in the LLVM compiler framework (Lattner and Adve, 2004) has been later presented as a proof of concept (Namjoshi, Tagliabue, and Zuck, 2013). It supports a limited set of instructions (enough to represent *while* programs over the integers) and a small set of transformations (simple constant propagation, dead code elimination and loop invariant code motion). The generated witnesses are checked for validity with the Z3 *Satisfiability Modulo Theories* (SMT) solver (Moura and Bjørner, 2008).

Since the proposed approach augments each optimization pass with a witness generator, it requires that the optimization procedures be known and could be examined. In fact, witness generation is not expected to be performed automatically: it assumes accesses to the optimization code and full knowledge of the optimization procedure. Furthermore, once the externally supplied invariants are consumed by a given optimization, and that no downstream optimizations will make use of these invariants, they will not be propagated any further in the compilation flow. Moreover, it is unclear how the proposed approach prevents optimizations that invalidate the semantics of invariants from happening. For instance, the witness relation only provides correspondences of variables referred in invariants before and after a given transformation, but does not actually forbid the transformation to remove such a referred variable but unused anywhere else in the program.

2.1.2 Functional Properties for Critical Real-time Systems

For hard real-time systems, it is not only crucial that the software computes the correct result, but also that this happens in a timely manner. One needs to determine the *Worst Case Execution Time* (WCET) of critical parts of the software to get an embedded system certified. WCET estimation has to be computed at the machine code level, because the timing of processor operations can only be obtained at this level. Information on program control flow is required to calculate WCET as accurately as possible. This information is called *flow facts* and takes the form of source code annotations about, for example, loop bound information (the maximum number of times a loop iterates, regardless of the program input), infeasible paths and program points that are mutually exclusive during the same run (*aiT*; Ballabriga et al., 2010). Interestingly, the source code annotations have also been used to express additional functional properties asserting some specific values of variables at a given program

point (Schommer et al., 2018; Eder et al., 2016), which might allow for more accurate WCET analysis.

Modern compilers translate high-level languages into binary code while applying hundreds of optimizations to deliver more performance. Some of them do not challenge the consistency of flow facts, whereas others radically modify the program control flow. As a result, it is usually very difficult to match the structure of the binary code with the original source code, and hence to port flow facts from high-level to low-level representations. Even when the structure of the binary and source code seem to match, there may be important changes of loop bound information, through optimizations such as loop unrolling or loop re-rolling. Using the flow facts obtained at the source code level or using best-effort methods for matching source code and binary code may be misleading. It is thus required to transform the flow facts of the program in accordance with the program compilation.

Various approaches have been proposed by the WCET community to address the issue (Kirner, 2003). Using the available debug information of the compiler can sometimes work as a simple mapping solution. In case of aggressive code optimizations, it is required to get additional support by the compiler. The simplest approach would be to transform the flow facts manually from the source code to the binary code (Li and Malik, 1997). This technique is simple to implement but hard to maintain in case of program updates and is potentially error-prone. Another approach would be to let the compiler generate a code optimization and transformation trace (Engblom, Ermedahl, and Altenbernd, 1998). However, this is too complex to support all types of optimizations. An alternative approach is to let the compiler do the mapping and generation of the correct flow facts of the binary code (Kirner and Puschner, 2001). This approach requires more compiler modifications but provides the most flexible support for code optimizations, thus will be examined more closely in the following.

Recently, a compiler framework (Li, Puaut, and Rohou, 2014), based on LLVM version 3.4, in which annotations on the source code are transformed into annotations on the binary level in the presence of compiler optimizations, has been proposed. For each LLVM optimization that modifies the program *Control Flow Graph* (CFG), a set of associated transformation rules (change rule, removal rule and addition rule) are defined in agreement to the CFG modifications. When the optimization pass is executed, the corresponding rules are applied to transform flow facts accordingly. As a consequence, this approach assumes deep understanding of the considered optimization algorithm in order to determine the transformation rules to be applied to flow fact annotations. Furthermore, only two types of flow facts are covered: loop bounds and additional flow constraints that are linear relations on execution counts of basic blocks. Therefore, other optimizations that do not modify the program CFG are not concerned; the mechanism is then not directly applicable to preserving functional properties in general.

Other work proposes an automatic mechanism to maintain annotations (expressing either flow facts or functional properties) from the source to the machine code level (Schommer et al., 2018). This is implemented in CompCert compiler, a formally verified and moderately optimizing compiler for C (Leroy, 2006; Leroy, 2009). A `__builtin_ais_annot` builtin is introduced as an extension of the C language into CompCert. Annotations via these builtins look like a call to a variadic function similar to `printf`: the first argument contains the annotation and is also a format string. It can contain format specifiers like %*here* or %e_i, where the former will be replaced with the absolute address of the annotation location in the final executable, while the

latter will be replaced with the location where the values of the i^{th} additional argument resides. CompCert treats `__builtin_ais_annot` as a call to an external function. No actual code is generated for the call, but the arguments of the builtin will be evaluated. The execution of the call is modeled as producing an observable event that includes the annotation string and the values of the arguments. CompCert's formal proof of semantic preservation guarantees that (1) annotations are not erased during compilation, unless they occurred in parts of the code that are unreachable during execution, and (2) they are neither reordered nor moved in the generated code, relatively to each other and relatively to other observable actions (such as external function calls and accesses to volatile variables). At the end of the compilation flow, CompCert collects all annotations contained in a compilation unit and stores them in an encoded form in a special section of the object file. Clearly, this approach seems to be a good solution to propagate functional properties from the source to the machine code level. Nevertheless, some issues may arise. The mechanism does not rule out the possibility that optimizations would move annotations around other non-observable computations. Although this is not the case for CompCert's optimizations, which are conservative and make no attempts to optimize around calls to unknown functions, it would be a serious problem when considering more aggressive optimizations from modern production compilers. For instance, classic loop optimizations would combine poorly with the annotation mechanism. First, most optimizations over loop nests, such as loop interchange or loop blocking, change the order in which iterations are performed. Therefore, they do not apply if the loop body can perform observable operations such as `__builtin_ais_annot`. Second, optimizations such as loop unrolling will modify the number of loop iterations (unrolling by a factor of k divides the number of iterations by k), while not being allowed to adjust the annotations, because this would change the observable behavior of the annotation according to the formal semantics. In short, by modeling program properties as observable events, the approach also implicitly creates barriers for many optimizations. Although our first proposed solution, described in Chapter 4, also relies on observable events to prevent optimizations from eliminating program properties, in Chapter 5, we show that properties need not be so restrictive (in terms of transformation constraints) in order to be preserved in optimizing compilation pipeline.

The ENTRA (Whole-Systems ENergy TRAnsparency) project Deliverable D2.1 (Eder et al., 2016) describes a similar mechanism to transfer flow facts and functional properties from source to machine code. Data and control flow properties are encoded as comments written as inline assembly expressions, relying on the compiler to preserve the local variables listed as inputs to the inline assembly. These expressions are declared as volatile I/O side-effecting to maintain their position in control flow relative to other code. As a result, the mechanism prevents certain optimizations from happening. Furthermore, this work does not attempt to formalize the preservation of properties as a correctness requirement, instead, the proposed mechanism relies on known and implementation-specific limitations of the compiler.

2.1.3 Functional Properties for Security Binary Analysis

As previously stated in Section 1.1.1, additional information about the program's expected behavior is required in order to carry out analysis assessing the correctness or robustness of the binary program. However, this problem has never been thoughtfully studied, since it is usually considered only as a secondary problem for security binary analysis frameworks. In general, program properties are given to the analysis

tools manually after a tedious and error-prone process of binary code inspection in order to determine the locations, in the executable binary, where the values of different variables occurred in the program properties are stored (Goubet et al., 2015; Bréjon et al., 2019; Laurent et al., 2019). Other work on automated process for detecting fault injection vulnerabilities in binaries has considered alternative solutions to such a manual approach (Given-Wilson et al., 2017). Properties to be validated and checked are expressed as assert statements in the source code. However, this is not satisfactory: assertions are only acceptable for executables in testing environments and are not suitable for use in production, as they generate run-time checks and thus consume extra execution time.

In short, the question of propagating additional properties down to machine code has always been considered as a side problem in security binary analysis community and thus has been subject to little research. However, we consider this as one of the main focuses of this thesis, and show that this allows for automated binary analysis (cf. Section 4.4.2.2).

2.2 Preserving Security Properties

Secure compilation is an emerging field aimed at developing compilers that preserve the security properties of the source programs they take as input in the target programs they produce as output. In the broadest sense, the goal of secure compilation research is to devise more secure compilation chains. Since there are many different ways to define "more secure", there are also many different notions of secure compilation. This section presents these different angles.

We first start by giving a broad classification of security properties widely-accepted by the community. Next, we introduce formal approaches to secure compilation with a focus on those that prove fully-abstract compilation, which has been the criterion adopted by much of the literature thus far. We then discuss existing work studying the interplay between side channels and secure compilers. Finally we present work that covers compiler implementations tailored to addressing fault injection attacks, but does not take a formal approach.

2.2.1 Classification of Security Properties

Computer security properties express what computer systems may and may not do. For example, a security property might stipulate that a system may not allow a user to read information that belongs to other users, or that a system may not delay too long in making a resource accessible to a user. Security properties have long been formulated in terms of a tripartite taxonomy: *confidentiality*, *integrity* and *availability*, which is hence called the *CIA taxonomy*:

- Confidentiality is the protection of information and system resources from unauthorized disclosure.
- Integrity is the protection of information and system resources from unauthorized modification.
- Availability is the protection of information and system resources from loss of use.

Although this is an intuitive categorization of security requirements, unfortunately, it is not supported by formal, mathematical theory: there is no formalization that simultaneously characterizes confidentiality, integrity and availability. Furthermore,

these elements are not completely orthogonal: for example, the requirement that a system be unable to read a value could be interpreted as confidentiality or unavailability of that value. Moreover, the CIA taxonomy provides little insight into how to enforce security requirements, because there is no verification methodology associated with any of the taxonomy's three categories.

This situation is similar to that of program verification in the 1970s. Many specific properties of interest had been identified, such as partial and total correctness, mutual exclusion, deadlock and starvation freedom, etc. But these properties were not all expressible in some unifying formalism, as they also are not orthogonal. These problems were addressed by the development of the theory of *trace properties* (Lamport, 1977; Alpern and Schneider, 1985). A trace is a sequence of execution states, and a property is a first-order predicate (i.e. a boolean function) that either holds or does not hold. Thus a trace property either holds or does not hold over an individual execution sequence.

Some security properties are expressible as trace properties. For example, consider the property stating that "the system may not write to the network before reading from the terminal". Formally, this is defined by the following set of traces:

$$\{t = s_0 s_1 \ldots \mid \neg(\forall i, j \in \mathbb{N} : i < j \wedge \mathit{isNetworkWrite}(s_i) \wedge \mathit{isTerminalRead}(s_j))\},$$

where $\mathit{isTerminalRead}(s)$ and $\mathit{isNetworkWrite}(s)$ are state predicates with the expected meaning.

Another example is the trace property "access control" requiring every operation to be consistent with its requestor's rights:

$$\{t = s_0 s_1 \ldots \mid (\forall i \in \mathbb{N} : \mathit{rights}(s_i) \subseteq \mathit{acm}(s_i)[\mathit{sub}(s_i), \mathit{obj}(s_i)])\},$$

where function $\mathit{acm}(s)$ yields the access control matrix in state s, function $\mathit{sub}(s)$ yields the subject who requested the operation that led to state s, function $\mathit{obj}(s)$ yields the object involved in that operation, and function $\mathit{rights}(s)$ yields the rights required for the operation to be allowed.

As another example, "guaranteed service" is a trace property requiring that every request for a service is eventually satisfied:

$$\{t = s_0 s_1 \ldots \mid (\forall i \in \mathbb{N} : \mathit{isReq}(s_i) \implies (\exists j > i : \mathit{isRespToReq}(s_j, s_i)))\},$$

where predicate $\mathit{isReq}(s)$ identifies whether a request is initiated in state s, and predicate $\mathit{isRespToReq}(s', s)$ identifies whether state s' completes the response to the request initiated in state s.

Unfortunately, trace properties are not expressive enough to capture a large class of security properties, since some important security properties cannot be expressed as properties of individual execution traces of a system. For example, "noninterference", as defined by Goguen and Meseguer (Goguen and Meseguer, 1982), is one of the most common examples of information-flow security properties. It is a confidentiality policy requiring that commands executed on behalf of users holding high clearances have no effect on system behavior observed by users holding low clearances. It is not a property of individual traces, because whether a trace is allowed by the policy depends on whether another trace (obtained by for example deleting command executions by high users) is also allowed. For another example, "stipulating a bound on mean response time over all executions" is an availability policy that cannot be specified as a property of individual traces, because the acceptability of delays in a trace depends on the magnitude of delays in all other traces. Rather,

they are properties of sets of execution traces, also known as *hyperproperties* (Clarkson and Schneider, 2010). Intuitively, a hyperproperty either holds or does not hold over a set of traces. Thus, a hyperproperty can be defined by a set of sets of traces, or equivalently a set of trace properties.

Let us now consider the example of information-flow security properties. These properties express restrictions on what information may be learned by users of a system. Users interact with systems by providing inputs and observing outputs. To model this interaction, define $ev(s)$ as the input or output event, if any, that occurs when a system transitions to state s. Assume that at most one event, input or output, can occur at each transition. For a trace t, let us extend this notation to $ev(t)$, denoting the sequence of events resulting from application of $ev(\cdot)$ to each state in trace t. Further assume that each user of a system is cleared either at confidentiality level L, representing low (public) information, or H, representing high (secret) information, and that each event is labeled with one of these confidentiality levels. Define $ev_L(t)$ to be the subsequence of low input and output events contained within $ev(t)$, and $ev_{H_{in}}(t)$ to be the subsequence of high input events contained within $ev(t)$. According to Goguen and Meseguer (Goguen and Meseguer, 1982), "noninterference" requires that commands issued by users holding high clearances be removable without affecting observations of users holding low clearances. Treating commands as inputs and observations as outputs, it can be modeled as a hyperproperty requiring a system to contain, for any trace t, a corresponding trace t' with no high inputs yet with the same low outputs as t. Formally, let TP denote the set of all trace properties, "noninterference" is defined by the set of sets of traces:

$$\{T \in TP \mid (\forall t \in T : (\forall t' \in T : ev_{H_{in}}(t') = \emptyset \wedge ev_{L_{out}}(t) = ev_{L_{out}}(t')))\}$$

A more generic variant of "noninterference" may require that commands issued by users holding high clearances be *different* without affecting observations of users holding low clearances. Still treating commands as inputs and observations as outputs, such security policy can be formally modeled as a set of sets of traces:

$$\{T \in TP \mid (\forall t \in T : (\forall t' \in T : ev_{H_{in}}(t) \neq ev_{H_{in}}(t') \wedge ev_{L_{out}}(t) = ev_{L_{out}}(t')))\}$$

As another example of hyperproperty, consider the "service level agreement" property which specifies acceptable performance of a system, using different statistics such as mean response time (mean time elapsed between a request and a response) or percentage uptime (percentage of time during which system is available to accept and service requests). These statistics can be used to define properties with respect to individual executions of a system or across all executions of a system. In the former case, "service level agreement" would be a trace property. For example, the property stating "the response time in each execution is less than 1 second" might not be satisfied by a system if there are executions in which some response times are much than 1 second. Yet if these executions are rare, then the system might still satisfy the property "the mean response time over all executions is less than 1 second". This latter "service level agreement" is not a trace property, but it is a hyperproperty:

$$\{T \in TP \mid mean\left(\bigcup_{t \in T} respTimes(t)\right) \leq 1\},$$

where function $mean(X)$ denotes the mean of a set X of real numbers, and $respTimes(t)$ denotes the set of response times (in seconds) from request/response events in trace t.

Many security properties have been classified as trace properties or hyperproperties (for even more examples, the interested reader is invited to read (Clarkson and Schneider, 2010)). However, the relation between this formulation and the CIA taxonomy is still an open question. In fact, the CIA taxonomy would seem to be orthogonal to trace properties and hyperproperties, which have the advantages of being formalized and providing a basis for expressing security properties. Furthermore, no verification methodology exists for CIA taxonomy, while there is such a methodology for trace properties (Alpern and Schneider, 1985), which is also generalized for hyperproperties (Clarkson and Schneider, 2010).

2.2.2 Fully-Abstract Compilation

Good high-level programming languages provide helpful abstractions for writing secure code in the form of type systems, module systems, encapsulation primitives and so forth. Unfortunately, most target languages do not offer the same security features as high-level source languages, and, certainly, plain untyped assembly languages such as those that run in commodity computers do not. Indeed, the security properties from the source language are generally not preserved when compiling a program and linking it with adversarial low-level code (e.g., a library or a legacy application). Linked low-level code that is malicious or compromised can for instance read and write the compiled program's data and code, jump to arbitrary instructions, or smash the stack, effectively violating security properties that the source program written in a high-level language had.

Secure compilation is the discipline that studies compilation schemes that preserve the security properties of source languages in their compiled, target-level counterparts, which is broad in scope. In the traditional sense, secure compilation is concerned with the security of partial programs, or *components*, and attackers are modeled as the environment such programs interact with. Partial programs are programs that do not implement all the functionality they require to operate. Instead, they are linked together with an environment, or *context*, that provides the missing functionality in order to create a runnable whole program. We may think of the context of a component as an attacker that interacts with the component, perhaps trying to learn some sensitive information (Abadi, 1999). A secure compilation chain protects source-level abstractions all the way down, ensuring that even an adversarial target-level context cannot break the security properties of a compiled program any more than some source-level context could.

The secure compilation literature contains several examples of source-level security properties that can be violated by target-level attackers (Patrignani, Ahmed, and Clarke, 2019). Attackers are often modeled as target-level programs, as this captures their ability to operate at that level. For example, consider an object-oriented source language that enforces private fields. Classes are thus allowed to store confidential data in private fields, making it inaccessible from other source-level code outside the class declaring the data. However, if this code gets compiled to a target language where memory locations are identified by natural numbers, such as an untyped assembly language, then the address where the confidential data is stored can be read by attackers. By dereferencing the number associated with the location of the data, attackers can violate the intended "confidentiality of private fields" property of the program. Another well-known security property that a secure compiler needs

to preserve is "integrity of local variables". Given a function defining a confidential local variable, then calling a callback function that was passed in as a parameter, the confidential variable is inaccessible to the code in the callback function at the source level. However if this function is compiled to a target language that can manipulate the call stack, the compiled callback function can access the confidential variable and change its value.

To protect compiled components from attacker contexts, secure compilation uses a variety of protection mechanisms, e.g., cryptographic primitives (Abadi, Fournet, and Gonthier, 2000; Corin et al., 2008), types (Ahmed and Blume, 2008), address space layout randomization (Abadi and Plotkin, 2012) protected module architectures (Agten et al., 2012; Patrignani et al., 2015), tagged architectures (Juglaret et al., 2017), etc. Once designed, researchers look to prove that such compilation schemes are indeed secure, i.e. they must be proven to conform to some criterion that implies secure compilation. A widely-used criterion for compiler security is *full abstraction* (Abadi, 1999).

Informally, a compiler is fully-abstract when it translates equivalent source-level components into equivalent target-level ones. Formally, a fully-abstract compiler *preserves* and *reflects* observational equivalence between source and target programs. Reflection of observational equivalence means that the compiler outputs target-level components that behave as their source-level counterparts, which is generally a consequence of the compiler correctness. Preservation of observational equivalence implies that the source-level abstractions in the generated target-level output are not violated by a target-level context. Notice that a fully-abstract compiler does not eliminate source-level security flaws. A fully-abstract compiler is conservative, as it introduces no more vulnerabilities at the target-level than the ones already exploitable at the source-level. Another point worth noting is, as previously discussed, the definition of full abstraction involves applying the compiler only to a component and not to the untrusted context in which it runs, which means that a fully-abstract compiler may choose to achieve its protection goals by introducing just a single barrier around the trusted part to protect it from the untrusted context (Juglaret et al., 2017; Patrignani et al., 2015; Patrignani, Ahmed, and Clarke, 2019).

Most existing work instantiates observational equivalence with *contextual equivalence*. Informally, two components P_1 and P_2 are contextually equivalent if they are interchangeable in any context without affecting the observable behavior of the program, i.e. they are *indistinguishable* just by looking at the outputs. Using contextual equivalence, only what can be observed by the context is of any relevance. Contexts can model malicious attackers that interoperate with the secure component, contextual equivalence can thus be used to model security properties of source code such as confidentiality or integrity (Abadi and Plotkin, 2012; Agten et al., 2012; Patrignani, Ahmed, and Clarke, 2019). Indeed, the confidentiality of a value means that it cannot be discerned by other code besides one allowed to manipulating it; a value v in a program P is thus confidential if P is contextually-equivalent to P' which is P with a different value for v. Similarly, integrity of a value means that it cannot be modified by other code besides one declaring it; a value v in a program P has integrity if P is contextually-equivalent to P' which is P where every interaction with other code is followed by a check that the value of v is the same as before the interaction.

In short, the emerging field of secure compilation provides powerful formally-proven means to protect high-level language abstractions in compiled code. However, one of the greatest challenges for secure compilation is the development of a secure compiler for a realistic programming language. The existing approaches show

promising results, but the application of secure compilation techniques to mainstream programming languages has not yet been achieved. In fact, fully-abstract compilation assumes that the source language itself is secure, so that it makes sense to define target-level security with respect to the semantics of the source language. More specifically, fully-abstract compilation is not well-suited for source languages with undefined behavior (e.g., C and LLVM IR): a possible approach consists in changing the criterion for secure compilation. For instance, a family of so-called robust criteria has been proposed recently (Abate et al., 2019). The authors thoroughly explore a large space of formal secure compilation criteria based on robust property preservation, i.e., the preservation of properties satisfied against arbitrary adversarial contexts. This includes various classes of trace properties such as safety (stating that some bad thing does not happen during execution) or liveness (stating that a good thing eventually happens during execution), of hyperproperties such as noninterference, and of relational hyperproperties (which are predicates on the behaviors of two or more programs) such as trace equivalence (which requires that two programs produce the same set of traces). In fact, in various deterministic settings, robust trace equivalence preservation coincides with preserving observational equivalence, the security-relevant part of full abstraction. This leads to many new secure compilation criteria, some of which are easier to practically achieve and prove than full abstraction, and some of which provide strictly stronger security guarantees. We refer the reader to the original survey for a detailed description of all these criteria (Abate et al., 2019).

In order to attain fully-abstract compilation, compilers often insert dynamic checks in the generated code to detect target-level contexts that interact with compiled code in ways that are impossible in the source language and respond to such interactions securely, often by halting the execution. Unfortunately, fully-abstract compilation is a very strong property, which preserves *all* source-level abstractions. This means that the kind of protections one has to put in place for preserving observational equivalence will likely be an overkill. Consider a password manager written in an object-oriented language, shown in the code snippet below, that is compiled to an assembly-like language.

```
1 private db: Database;
2
3 public testPwd(user: Char[8], pwd: BitString) : Bool {
4   if (db.contains(user))
5     return db.get(user).getPassword() == pwd;
6 }
```

The source program exports the function `testPwd` to check whether a user's stored password matches a given password `pwd`. The stored password is in a local database, which is represented by a piece of local state in the variable `db`. The details of `db` are not important here, but the database is marked private, so it is not directly accessible to the context of this program in the source language.

A fully-abstract compiler for the program above must generate code that checks that the arguments passed to `testPwd` by the context are of the right type (Patrignani and Garg, 2019). The code expects an array of characters of length 8. A parameter of a different type cannot be passed in the source, so it must also be prevented in the target. Since the target is untyped, code must be inserted to check the argument. Specifically, a fully abstract compiler will generate code similar to the following (assuming that the arrays are passed as pointers and the base address is stored in register `r0`).

```
label testPwd:
  for i = 0; i < 8; ++i
    load the memory word stored at address r0 + i into r1
    test that r1 is a valid char encoding
  ...
```

Basically, this code dynamically checks that the first argument is a character array of length 8 because a type mismatch could lead to a violation of fully-abstract compilation. Such a check can be very inefficient when the length is very long. This example shows that stronger secure compilation criteria are harder (or even impossible) to achieve efficiently. Even when efficiency is not a concern, stronger secure compilation criteria are still harder to prove (Agten et al., 2012; Patrignani et al., 2015). As a result, a new research path is recently envisioned. As mentioned previously, this consists in developing new secure compilation criteria other than full abstraction that fine-tune the checks inserted by the compiler so that they only affect code security (Abate et al., 2019; Patrignani and Garg, 2019; Patrignani and Garg, 2017).

Furthermore, research in secure compilation has primarily focused on identifying and formalizing secure compilation criteria, and on developing effective formal verification techniques, but has rarely considered the problem of securing compiler optimizations (Patrignani, Ahmed, and Clarke, 2019). Investigating the interaction between code optimization and security has been recently proposed but has not been thoroughly carried out (D'Silva, Payer, and Song, 2015). As a result, it is an open problem to determine how security properties preserved by fully-abstract compilation would interact with various compiler optimizations. More specifically, studying which optimizations violate security properties, typically by removing the dynamic checks inserted by a secure compilation scheme, is an interesting research path for secure compiler development.

2.2.3 Secure Compilation Against Side-Channel Attacks

Most existing work on secure compilation proves (or assumes) compiler correctness and proves compiler full abstraction using the notion of contextual equivalence described in the previous subsection. However, it has been shown that some intuitive and interesting security properties are not necessarily preserved by such a compiler (Patrignani and Garg, 2017). In fact, contextual equivalence has shown its limitations both in the attacks it can express, the efficiency of the code it generates and the complexity it introduces in proofs. Side-channel attacks, which abuse information gained from observing physical behaviors of computer systems, cannot be expressed with contextual equivalence. These attacks are generally disregarded by secure compilation techniques (Patrignani, Ahmed, and Clarke, 2019). Indeed, secure compilers ensure that the compiled programs behave as intended by the source programs according to the high-level language specifications, while these attacks are based on behaviors not defined in the abstractions of language standards, e.g. power consumption, execution time or cache behavior. Protections against side-channel attacks seek to eliminate the information leakage from these physical channels. As a result, the security properties associated with these protections cannot be understood by standard compilers, and may be removed or compromised during program transformations.

There is a large body of research and engineering work targeting at closing down side-channels. There are two security properties that are particularly of interest:

preserving the secret independence guarantees of the source code (also known as *constant-time programming*) and preserving the erasure of secret data.

2.2.3.1 Cryptographic Constant-Time Preservation

In order to close the timing side-channel, security engineers follow a very strict programming discipline called constant-time programming. This name is a bit of a misnomer, as they do not intend to make the programs they write literally constant-time, but constant-time *with regards to* secrets, i.e., program's execution times do not depend on secrets. This is achieved by ensuring that neither control-flow (branchings) nor memory access pattern of the programs depends on secrets, often by leveraging bitwise operators. However, it has been reported that this style of code recommended by cryptographers, when compiled for some architectures, produces binary code that is actually not constant-time (Simon, Chisnall, and Anderson, 2018; Kaufmann et al., 2016).

Recent work has studied the question of whether the code generated by a compiler for a constant-time source program is itself provably constant-time (Barthe et al., 2019). The authors address the challenges of secure compilation from the specific angle of turning the formally-verified compiler CompCert into a formally-verified secure compiler with regards to the specific property of constant-time.

Specifically, they first enrich the semantics of CompCert intermediate representations with a formal notion of constant-time. Intuitively, a program P is said to be constant-time if two of its executions that differ only on their secret inputs induce equal leakage. The leakages of P are modeled as a list of atomic leakages, where an atomic leakage results from a single step of the program execution. An atomic leakage is either the truth value of a condition, a pointer representing the address of either a memory access, or a called function. The traces of input/output events defined in CompCert are then enriched with leakages, represented as a new kind of events. This way, the authors benefit from all the existing definitions and lemmas about traces that they directly reuse in their proofs.

Given this semantics extension, they then identify transformation passes of CompCert that do not preserve their constant-time policy. In particular, it has been reported that the "instruction selection" pass introduced conditional branches in compiled code. The authors then modify CompCert so that the compilation does not introduce conditional branches anymore by defining a new architecture-dependent selection operation. The modified compiler only targets the x86 architecture, the introduced selection operation is thus compiled down to the `cmov` instruction.

Finally, the authors design new proof techniques and apply these, together with the proof structure already present in CompCert, to prove that 17 out of 20 compiler passes in the modified CompCert preserve constant-time.

In short, this work specifies a machine-checkable proof that a mildly modified version of a moderately optimizing compiler preserves a specific security property of constant-time. The authors extensively study *every* compiler optimization pass in order to identify ones that do not preserve constant-time and adjust them accordingly, then formally prove that the modified version does indeed preserve constant-time. However, this approach does not scale to more aggressively-optimizing compilers used in industrial settings, which may contain up to hundreds of optimizations. Note that this open problem of designing a scalable property preservation approach is one of the main motivations of this thesis.

2.2.3.2 Secret Erasure Preservation

The problem of preserving the secret erasure property has also been studied using a similar approach to the work presented above (Besson, Dang, and Jensen, 2018). The overall goal is to prevent information leaks from being introduced by the compilation process. In other words, compiled code should be no more vulnerable to passive side-channel attacks than the source code. Such side channels correspond to an attacker who is granted physical memory access at specific observation points. At the semantic level, side channels can be modeled using a leakage function exposing a partial view of the program state to the attacker. The semantics of a program P is given by a one-step deterministic transition relation $\rightarrow$ which is either a silent transition, or a leaked transition indicating that the end state of the transition is leaked to the attacker. From an initial memory m_0, a trace of P is given by a sequence of memories m_0, m_1, ... m_n such that for every $i < n$, we have $m_i \rightarrow_i m_{i+1}$.

Given such a trace, the observation of the attacker is given by the sub-trace of leaked memories, i.e. those memories resulting from a leaked transition. The secure transformation of a source program into a target program, called *Information Flow Preserving* (IFP) transformation, is then defined as follows. A source program P is at least as vulnerable as a target program P' if any observation O' of a target trace of P' can always be matched by an observation O of a source trace of P such that O leaks more information than O'. The notion of *matching observation* is formalized by a function mapping observations from source to target trace.

Given the formal definition of IFP transformation, the authors identify a set of sufficient conditions to prove that a transformation is IFP, and use them to study two compiler transformations which are not *a priori* IFP: "dead store elimination" and "register allocation". They show how to make these two transformations IFP and prove them sound using two different proof techniques.

Dead Store Elimination: Dead stores are typically identified using a liveness analysis which computes, for each program point, an over-approximation of *live* variables, i.e. variables that are necessary to compute the program result. Dually, *dead* variables are those that are not live, and a dead store is a write to a memory address where the value stored at this address is dead. A classic "dead store elimination" performs a liveness analysis and removes dead stores. As shown in the example of Listing 1.2 from Section 1.1.2, when a dead store has the security purpose of erasing sensitive values from memory, "dead store elimination" introduces sensitive information leaks. In order to make the transformation IFP, it is sufficient to slightly modify the initial condition of the liveness analysis by imposing that, at the end of the program, every address is live. The effect of this is that only dead stores that are shadowed by a following store at the same address can be safely removed. In other words, for a given address, the last store needs to be retained (whether it is dead or not).

Register Allocation: As for this transformation which compiles source programs using an unbounded number of variables into target programs using a limited number of hardware registers, the authors resort to an *a posteriori* approach using a certified validator which checks the IFP condition for a particular program and its "register allocation" transformation. In fact, the resource allocation task may be impossible due to a shortage of registers. In that case, a register may be *spilled* in the function stack frame, i.e. its content copied, for later reuse. Furthermore, after the last use of a spilled register, a conventional "register allocation" algorithm has no reason

to explicitly erase the stack location. This clearly breaks the IFP property, as it introduces an information leak through the duplicated value spilled in the stack. To ensure that "register allocation" is IFP, a variation of the translation validation approach is proposed. The condition says that for every non-constant address A' of the transformed program P', there must exist a matching address A in the original program P. A constant address is one which returns the same value for every execution of the program, while two addresses are in matching correspondence if they contain the same value for every execution of the program. Conventional "register allocation" pass keeps the memory mostly unchanged except for a limited number of registers and spilled locations. The set of addresses is then partitioned into a subset of named addresses affected by "register allocation" and a subset of unnamed addresses untouched by the transformation. The validator thus needs to establish a mapping between named addresses, while only needs to check that unnamed addresses have the exact same value before and after the transformation. Then, the missing mappings of the validator which indicate the information flow leaks allows recovering from certain validation failures. Closing these leaks can be done, during a post-treatment, by inserting erasure instructions which zero all those registers and spilled locations just before the attacker observation.

The implementation of these two IFP transformations are later experimented within the CompCert compiler (Besson, Dang, and Jensen, 2019). In short, the proposed approach provides a means for proving preservation of information-flow, and also illustrates how to modify non-IFP transformations of a moderately optimizing compiler accordingly so that they preserve information-flow. Similarly to work presented in Section 2.2.3.1, it is unclear how the same approach can be used when scaling up to more aggressively-optimizing compilation flow, as it requires identifying all non-IFP passes and potentially modifying each of them.

2.2.3.3 Preventing Side-Channel During Compilation

It is worth noting that these studies all try to answer the question of preserving security property during compilation. Indeed, the security of applications is devised by the developers at the source level, which needs to be preserved through compilation flow, all the way down to the binary code. Nonetheless, this is not the only approach to secure compilation. Another idea consists in enforcing the program security during compilation. Regardless of the input programs, the compiler will always generate programs that satisfy a defined security policy. For instance, to mitigate the problem of secret erasure caused by "dead store elimination" described previously, a recent work proposes a modified version of the clang compiler that automatically erases all registers and the whole stack frame after returning from a sensitive function (Simon, Chisnall, and Anderson, 2018). The same work also demonstrates a solution to the problem of the compiler generating conditional branches for the constant-time selection operation. The authors propose another variant of clang that always compiles constant-time selections to conditional move instructions (when available in the target architecture, such as IA32, x86-64 or various ARM architectures) or to sequences of bitwise operations implementing the selection without conditional branches.

As previously stated, this falls into the category of security enforcement during compilation rather than security preservation: it focuses on target-level enforcement of specific policies but without any connection to source-level properties. Furthermore, the proposed implementation of constant-time selection happens at the start

of the compiler back-end, so it cannot avoid downstream optimizations of the back-end. Theoretically, such an optimization might turn constant-time selections into conditional branches. Although the presented benchmarks do not show any example of this, the approach does not provide any formal statement of correctness. As for the proposed solution to secure secret erasure, the instrumentation is implemented in the compiler back-end after other back-end optimizations have taken place, which gives stronger guarantees that the erasure is not altered. Nonetheless, systematically erasing the whole stack frame and registers used in a sensitive function may not be the most efficient approach. Intuitively, erasing only stack slots and registers containing a copy of the secret data, annotated by the programmer in the source code, before returning from the sensitive function would be more reasonable. However, this requires that the compiler be able to correctly propagate the source annotation beyond "register allocation". Yet, practical experiments are needed to assess the effect of each approach on the quality of the generated code.

2.2.4 Secure Compilation Against Fault Injection Attacks

Other than side-channel attacks, sensitive data is also subject to fault injection attacks. Secure elements widely used in smart phones or payment systems are common targets of such attacks. Furthermore, with the emergence of the *Internet of Things* (IoT), personal data may be handled by a huge variety of devices, which are also subject to fault attacks. These attacks aim at disrupting the execution of applications to extract sensitive information or grant restricted access permissions (Yuce, Schaumont, and Witteman, 2018; Moro et al., 2013; Timmers, Spruyt, and Witteman, 2016; Bar-El et al., 2004). As a reaction, countermeasures have been proposed to protect embedded systems against faults. In general, all of these approaches rely on some form of spatial or temporal redundancy (Witteman, 2018). Unfortunately, compilers have always been designed to optimize programs, mostly by eliminating redundancy, which makes it notoriously hard to preserve these countermeasures through compilation (Hillebold, 2014). Indeed, code hardening usually suffers from compiler optimizations damaging or even completely removing the protections. Let us now present these approaches and demonstrate the issue.

2.2.4.1 Data Integrity Protection

The first example is a countermeasure ensuring data integrity, implemented in the LLVM compiler (Hillebold, 2014). New transformation passes are added to the compilation pipeline to implement simple duplication, which aims at storing each variable multiple times and performing computations on redundant data, and complementary redundancy, which aims at storing the binary inverse of the original values and performing complementary instructions on inverted data. As expected, the redundancy introduced by these transformations is not preserved by downstream optimization passes, notably in the compiler back-end. The authors thus have to explicitly deactivate these passes which remove redundant computations. Furthermore, loading redundant constants is not straightforward. A given constant is only loaded once using a `move` instruction, and the inverse constant is also generated from the same source as the original constant by inverting the latter. This thus results in detected successful attacks in their experiments for both duplication schemes, and closing this leak against fault injections would require significant modification to the compiler back-end, notably the instruction selector.

2.2.4.2 Protecting Against Instruction Skips

The second example is an implementation in the LLVM compiler (Barry, Couroussé, and Robisson, 2016) of a formally-verified duplication scheme that protects low-level code against faults inducing instruction skips (Moro et al., 2014), targeting exclusively ARMv7-M/Thumb2 instruction set. The protection scheme consists in transforming all machine instructions into a semantically-equivalent sequence of idempotent instructions, which will then be duplicated. An idempotent instruction is one that can be re-executed without changing the resulting state of the program. This means that if one of the duplicated instructions is faulted, resulting in an instruction skip, the program is still correct. The authors modify the instruction selector and register allocator to generate idempotent instructions. For other instructions that need special treatments such as non-idempotent variant of store, branch-and-link, or if-then block, new transformation passes have been inserted to turn them into sequences of idempotent instructions. Finally, all instructions are duplicated. Unlike the example from Section 2.2.4.1, instruction duplication is performed as late as possible in the compiler back-end instead of in the compilation optimizer pipeline, which gives stronger guarantees that the duplication is not removed.

2.2.4.3 Loop Protection

Loops in sensitive code are important targets of fault attacks. For example, it has been shown that corrupting `memcpy` during the initialization of an embedded system may allow an attacker to escalate privileges and execute arbitrary code (Timmers, Spruyt, and Witteman, 2016). Other work also highlighted the need to protect the iteration count of the `memcmp`-like PIN authentication (Dureuil et al., 2016). There has been recent work to automatically harden sensitive loops at compile-time (Proy et al., 2017). The authors state that a sensitive loop must always perform the expected number of iterations and take the right exit, otherwise an attack must be reported. They implement, in the LLVM compiler, a generic loop hardening scheme based on duplication of termination conditions and of the computations involved in the evaluation of such conditions. The hardening is implemented as a transformation pass, run at the start of the compiler back-end, while still operating on the target-independent IR of the compiler. This positioning of the pass requires a careful analysis of the compilation flow, considering both the dependence on upstream optimization passes providing the necessary information to implement the countermeasure and the interferences with downstream passes. Nonetheless, the back-end optimizations do alter the countermeasure scheme to some extent. As explained in Section 2.2.4.1, the loading of constants cannot automatically be duplicated. Resolving this requires modification to the compiler instruction selector. Furthermore, addresses of global variables are not available at IR level, hence it cannot be duplicated when loading the value of the variable. These interferences with back-end passes result in a small amount of harmful undetected faults, which again highlights the challenge of preserving security protections in an optimizing compilation flow.

2.3 Discussion

This chapter presents the existing work on preserving program properties of the source program, through compilation, down to machine code. These properties can

be used to enhance the compilation process—from augmenting optimizations to securing generated programs—or to support automated analysis of binary code. Different approaches have been separately proposed to address the issue, considering a single application field. Notably, this is an active research problem in the secure compilation community.

More specifically, for critical real-time systems, the community has proposed diverse mechanisms to propagate source-level properties describing flow facts—required to perform WCET estimation at binary level—such as loop bound information or infeasible paths. A direction consists in defining transformation rules associated to every optimization that modifies the program CFG, which will update the flow facts in concert with CFG modifications. Other work proposes to represent property as a builtin that includes the values of variables referred in the property, and is considered as an observable event by the compiler, so that it is not erased during compilation.

As for the problem of preserving source-level security properties down to generated binary, secure compilation community has come up with various research directions. Well-studied notion of fully abstract compilation—which preserves observational equivalence under an attacker model where attackers are target-level contexts—provides a partial answer, ensuring the absence of target-level attacks like control flow hijacks for example. Other work studies classes of security properties other than observational equivalences, which may provide strictly stronger security guarantees than full abstraction, or may be easier to practically achieve and prove.

Instead of preserving classes of security properties, another direction of research seeks to prove that a compiler preserves security properties in very specific scenarios, such as constant-timeness or information-flow preservation. To this end, a natural approach consists in leveraging pre-existing proof technique designed for proving compiler correctness. As a result, formally-verified compiler such as CompCert is usually used to illustrate the approach, which can be basically broken down into three steps: (1) identify the compiler passes that do not preserve the considered security property and adjust them accordingly; (2) formalize the considered security for compiler intermediate representations; and (3) prove that all (modified) passes preserve the property, according to the formalization developed in the previous step.

Lastly, there is other work that cover compiler implementations tailored to addressing specific attacks—notably fault injection—but which does not take a formal approach. More specifically, countermeasure schemes against specific attack models are devised and implemented in modern optimizing compilers, such that code hardening is automatically performed at compilation time.

Based on our observation, we have identified two major issues that need to be addressed, which have lead to the research direction that we took in the work described in this thesis.

First, there is a need for functional properties when carrying out analysis on executable binaries. These properties generally take the form of first-order predicate involving variables or structures defined in the source program and should be evaluated at a specific program point. Clearly, directly expressing these properties at binary level is not ideal, because it is difficult for engineers to reason in machine code and thus necessitates program reverse engineering. The alternative approach is to introduce the properties at the source level and propagate them through the compilation process. However, this is only thoroughly explored for conveying flow facts required to compute WCET for hard real-time systems, and no existing work has really studied the problem for functional properties needed by other types of analysis. As a result, our first axis of research aims at designing a compilation mechanism to

propagate and maintain source-level functional properties down to executable binary. Moreover, we strive to not simply propose a proof of concept—which may be reused by subsequent research projects—but conceive a well-engineered solution to contribute to a broader community. Therefore, we target source program written in widely-used programming languages such as C, and we would like to implement our proposed mechanism in widely-used production compilers such as LLVM or gcc (Stallman and Community, 2009), which makes it infeasible to consider each optimization individually. In other words, we deliberately seek to devise a generic, optimization-agnostic functional property preservation mechanism. This also implies that handling future optimization passes introduced to the compilation flow would be effortless, or even requires no modification at all. This axis will be discussed in Chapter 4.

As for security properties, and perhaps surprisingly, very little work has considered the problem of carrying these along modern optimizing compilation chains used in industrial settings. More specifically, it has been reported that these compilers did not understand the intentions of software-based countermeasures—generally introduced to the program at the source level—and optimized them away as a result; nevertheless, to the extent of our knowledge, no previous work has proposed a mechanism to prevent aggressively-optimizing compilers from invalidating or eliminating security countermeasures. As a consequence, the second goal of the work described in this thesis is to devise such a mechanism. Convinced that there is no "one-size-fits-all" solution for preserving different countermeasures against different types of attacks that ensure security properties of different natures, we address the challenges of secure compilation from a very specific angle of approach. First, given a protection scheme, we identify the security protection-derived properties, which are defined such that preserving them also implies the preservation of the protection. We then devise a careful encoding of these as functional properties, so that we can leverage the same functional property preservation mechanism to conserve security countermeasure, and this also in an optimization-agnostic manner. This will be described in Section 4.4.3, and further detailed in Chapter 5.

The functional properties that we consider can be classified as trace properties. For instance, the property associated with the secret buffer erasure protection scheme presented in Section 1.1.2 can be defined as the following set of traces:

$$\{t = s_0 s_1 \ldots \mid (\exists i \in \mathbb{N} : \mathit{isVerifyPoint}(s_i) \wedge \mathit{isBufferZeroed}(s_i, \mathit{secret}))\},$$

where $\mathit{isVerifyPoint}(s)$ is a state predicate indicating whether the state s is one denoted by the label `verify_point` in the program source, and $\mathit{isBufferZeroed}(buf, s)$ is a state predicate identifying whether the whole buffer buf is zeroed in state s.

However, as stated in Section 1.2, the particularity of the properties that we consider is that they are *not* part of the program semantics, but are kept external to the code itself. Therefore, the biggest challenge lies in the fact that compilers do not know about these properties and certainly not how to maintain them along program transformations. Furthermore, it is significantly harder to address this issue, notably when we aim at targeting an aggressively-optimizing compilation flow. For example, if a static variable is mentioned by name in a property but unused anywhere else, optimizations will remove this variable and thus make the property meaningless. We will detail our proposed solution to the problem of preserving functional properties, as well as its implementation in a modern optimizing compiler in Chapter 4.

Chapter 3

Tools and Security Use-Cases

In this chapter, we present the important toolchains based on which we will develop our property-preserving compilation frameworks, which will be described in Chapter 4 and Chapter 5. Moreover, we further describe a set of security use-cases that we will use to illustrate the application of our solutions to address the problem of preserving source-level security countermeasures during compilation.

Recalling the goal of this thesis is to propose an property- and optimization-agnostic approach for preserving program properties throughout optimizing compilation flow, so that it can be implemented in an aggressively-optimizing compiler and be made available to a broad community. Due to its widespread use in academic and industrial settings, as well as its modularity and high reusability, we decided to demonstrate our approach on the LLVM compiler toolchain, which will be presented in Section 3.1.

Let us remind that the goal of our property-preserving compilation is twofold: it provides (1) additional information—usually expressed as functional properties—required by various analyses performed on the executable binary, and (2) a reliable means to ensure the presence, in the machine code, of diverse security protections, by preserving protection-derived properties. As explained in Section 1.1, both types of properties are not actually part of the program functional semantics and need to be kept external to the code itself. Furthermore, functional properties are meant to be evaluated by binary analysis tools, it is mandatory to define a means for these tools to retrieve the properties from the executable binary. As for protection-derived properties, it is also useful to have some meta-information describing them at the binary level. For instance, the meta-information may communicate how (e.g. which variables need to be evaluated and how to look up their values) and when (e.g. at which machine address during program execution) to evaluate these properties at execution time, for validation purposes (this will be detailed further in Section 4.4.3 and Section 5.5.2). Clearly, this meta-information is not part of the program and have to be externally attached to the machine code. As a consequence, we need to define an interface to retrieve the extra information (i.e. properties for binary analysis or meta-information describing protection-derived properties) from the executable binary.

Properties usually involve variables or structures defined by the programmers in the source program. Thus, consumers of these properties at the binary level need to figure out the corresponding representation of the variables or structures in the executable binaries in order to carry out the analysis of the program, or to verify these program properties. Interestingly, this problem shares some similarities with the debugging process.

As a compiler reads and parses the source of a program, it collects a variety of information about the program, such as the line numbers where a variable or function is declared or used, which will be useful later when the program is debugged.

Indeed, the task of a debugger is to provide the programmer with a view of the executing program in an as natural and understandable fashion as possible. This means that the debugger has to essentially reverse much of the compiler's transformations, converting the program's data and state back into the terms that the programmer originally used in the program's source. To make this possible, debug information is generated by the compiler for the purpose of communicating source location, type and variable information to the debugger. It is thus very reasonable to extend the debug information to represent properties in the binary. To this end, we use the *Debugging With Attributed Record Formats* (DWARF) debug data format (Eager, 2007; DWARF Debugging Information Format Commitee, 2017) that provides an easily extensible description of how a program is translated into executable code: a debugger (or any consumer of the debug data) can recognize and ignore an extension, even if it might not understand its meaning. The DWARF debug format will be presented in Section 3.2.

Finally, we describe in Section 3.3 our selected security use-cases demonstrating different protection schemes against physical attacks (side-channel attacks and fault injection attacks) that we later use to illustrate our approaches and validate their implementations.

3.1 LLVM Compilation Infrastructure

In this section, we first start by presenting the overall architecture of traditional compilers, which is also adopted by LLVM. We then describe the most important component of the framework—the LLVM IR—as well as some of its key features needed for our implementations, namely intrinsic functions and metadata. Finally, we detail the LLVM code generator, presenting different compilation phases transforming the IR into machine code.

3.1.1 LLVM Overview

This subsection discusses some of the design decisions that shaped the LLVM project, which consists of several libraries and tools that, together, make a large compiler infrastructure.

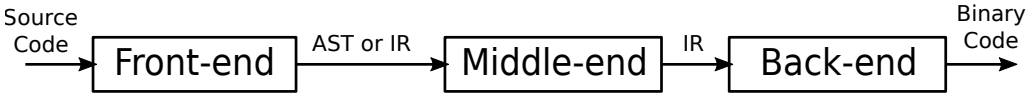

FIGURE 3.1: Components of a three-phase compiler.

The most popular design for a traditional compiler (like most C compilers) is the three-phase design—shown in Figure 3.1—whose major components are the front-end, the middle-end (also known as the optimizer) and the back-end (also known as the code generator).

- **Front-end**: this is the compiler step that parses source code, checking it for errors, and builds a language-specific *Abstract Syntax Tree* (AST) to represent the input code. The AST may be optionally converted to a new representation for optimizations.
- **Middle-end**: this is responsible for doing a broad variety of transformations to try to improve the code's running time, such as eliminating redundant computations, and is usually independent of source language and target architecture.

- **Back-end**: this is the step that is responsible for code generation: it maps the code onto the target instruction set. In addition, it also tries to generate target code that takes advantage of specific features of the supported architecture.

With this design, porting the compiler to support a new source language requires implementing a new front-end, but the existing optimizer and back-end can be reused. As a result, the LLVM infrastructure also follows this design.

More specifically, the LLVM front-end first translates high level programming languages into an internal code representation known as the LLVM IR. We will consider in the following the clang project—the official LLVM frontend for C, C++ and Objective-C.

Next, the LLVM middle-end provides a modern source- and target-independent optimizer: it takes LLVM IR as input, runs the specified optimizations on it, and then outputs the optimized LLVM IR or the analysis results. In LLVM, the optimizer is organized as a pipeline of distinct *LLVM passes* such as the inliner—which substitutes the body of a function into call sites, instruction combining, CFG simplifying, expression reassociation, loop invariant code motion, etc. Depending on the optimization level, different passes are run: for instance, at `-O0` (no optimization), the compiler runs no passes, while at `-O3` (all optimizations), it runs a series of 256 passes (as of LLVM 12.0.0). These optimizations are usually composed of analysis and transformation passes: the former recognizes optimization opportunities and generates the necessary data structures describing the analysis results that can later be consumed by the latter.

Lastly, the LLVM code generator converts LLVM IR to target-specific assembly code or object code binaries. Instruction selection, instruction scheduling, register allocation, peephole optimizations and target-specific optimizations/transformations all belong to the code generator.

3.1.2 LLVM Intermediate Representation

The LLVM IR is the backbone that connects front-ends and back-ends, allowing LLVM to parse multiple source languages and generate code to multiple targets. Front-ends produce the IR, while back-ends consume it. The IR is also the point where the majority of LLVM target-independent optimizations takes place.

An IR in SSA: The LLVM IR uses the *Static Single Assignment* (SSA) form: there is no value that is reassigned, each value has only a single assignment that defines it. Each use of a value can immediately be traced back to the sole LLVM IR instruction responsible for its definition. This has an immense value to simplify optimizations, owing to the trivial use-def chains that the SSA form creates, that is, the list of definitions that reaches a user. As a consequence, it has an infinite number of registers.

IR extension using intrinsics: LLVM supports the notion of intrinsic functions[1], which have well known names and semantics and are required to follow certain restrictions. Overall, these intrinsics are used to convey high-level semantics to the compiler, while representing an extension mechanism for the LLVM IR that does not require changing all of the transformations in LLVM when adding to the IR. Examples of intrinsic functions range from supports for important standard C library functions such as `sqrt()` or `memcpy()`, to supports for garbage collection, exception

[1] https://llvm.org/docs/LangRef.html#intrinsic-functions

handling or even vector horizontal reduction. The compiler knows how to best implement the functionality in the most optimized way for these functions by replacing them with a set of machine instructions for a particular back-end. Intrinsics require the compiler to follow additional rules while transforming the program. These rules are communicated to the compiler via the *function attributes* which specify the intrinsic's behavior with regards to the program mutable state (such as memory, control registers, other side-effects, etc.).

3.1.3 LLVM Metadata

LLVM IR allows metadata to be attached to different IR entities in the program such as functions or instructions in order to convey extra information about the code to the optimizer[2]. In fact, this metadata is typically used to influence optimization passes. For example, metadata is added to the IR to describe a type system of a higher level language, and this can be used to implement C/C++ strict type aliasing rules. Another example of metadata use is to suggest an unroll factor to the loop unroller using loop identifier metadata.

However, since all of these typical uses of LLVM metadata are to provide optional information to the optimizer, there does not currently exist any mechanism to maintain and propagate metadata in the back-end. The only exception to this is debug information metadata. Debug information communicates source location information, type information and variable information to the debugger[3]. This information is not used during the execution of program and does not result in executable code in the object file, but the back-end uses it to produce DWARF information. In other words, debug information can be seen as a sort of side channel from the front-end to the DWARF emitter in the back-end.

The idea of the LLVM debug information is to provide an LLVM user a relationship between generated code and the original program source code. To do this, most of the debug information, such as descriptors for types, for variables, for functions, for source files, . . . , is inserted by the front-end in the form of LLVM metadata. LLVM optimizations are upgraded to be aware of debug information, allowing them to update the debug information as they perform aggressive optimizations.

A subtle point about metadata (debug information or not) is that it was designed to be ignored by the optimizer and can be discarded without affecting correctness. This implies that metadata-based debug information is a best-effort feature: if metadata is dropped by optimizations, it just means that debug information quality is reduced, it does not invalidate the debug information itself; which is an acceptable (but should not be) behavior when debugging optimized code.

3.1.4 LLVM Code Generator

The back-end is comprised of the set of code generation analysis and transformation passes that converts the LLVM IR into the machine code for a specified target—either in assembly form or in binary object code format. There are several steps involved in transforming the LLVM IR into target code. The IR is converted to a back-end-friendly representation of instructions, functions and globals. This representation changes as the program progresses through the back-end phases and gets closer to the actual target instructions. This translation pipeline is composed of different phases of the back-end:

[2]https://llvm.org/docs/LangRef.html#metadata
[3]https://llvm.org/docs/SourceLevelDebugging.html

- **Instruction Selection**: this phase converts the three-address structure of the LLVM IR to a *Directed Acyclic Graph* (DAG) form. The DAG nodes typically represent instructions, while the edges encode a data-flow dependence among them. The transformation to DAG is important to allow the LLVM back-end to employ tree-based pattern-matching instruction selection algorithms. By the end of this phase, the DAG has all of its LLVM IR nodes converted to target-machine nodes, that is, nodes that represent machine instructions rather than LLVM instructions.

- **Scheduling and MIR Formation**: this phase takes the DAG of target instructions produced by the instruction selection phase, determines an ordering of the instructions while trying to explore instruction-level parallelism as much as possible, then emits the instructions with that ordering. The output program of this phase is represented using a machine specific representation called *Machine Intermediate Representation* (MIR). The LLVM MIR represents instructions in their most abstract form: an opcode and a series of operands. This is designed to support both an SSA representation for machine code, as well as a register-allocated, non-SSA form.

- **SSA-based Machine Code Optimizations**: this optional phase consists of a series of machine code optimizations, such as machine-level dead code elimination or instruction combining, that operate on the SSA form produced by the instruction selector.

- **Register Allocation**: this phase transforms an infinite set of virtual register references into a finite (possibly small) set of physical target-specific registers. If the number of physical registers is not enough to accommodate all the virtual registers, some of them will have to be temporary stored into the stack. This process is called "spilling" the registers.

- **Late Machine Code Optimizations**: this stage is responsible for implementing optimizations that operate on final machine code, such as peephole optimizations and notably the post-register-allocation instruction scheduling. In fact, since real register information is now available, the presence of extra hazards and delays associated with certain types of registers can be used to improve the instruction order.

- **Code Emission**: the final phase actually emits the code for the program, as well as its debug information, either in the target assembler format or in machine code.

In addition, target implementations can insert arbitrary target-specific passes into the flow.

Intrinsics from LLVM IR are generally lowered into pseudo-instructions. These are special MIR instructions that do not have machine encoding information and must be expanded, at the latest, before code emission. This allows for generation of optimized machine code implementing the intrinsics' semantics. Just as for intrinsics in the IR, the high-level semantics as well as the behavior with regards to the program mutable state of pseudo-instructions can be controlled via various *instruction attributes*. These define extra constraints that need to be respected by compiler optimizations when transforming programs containing the pseudo-instructions.

3.2 DWARF debug format

This section describes the important features of the DWARF standard that we will need for our extension to describe program properties at the binary level, while the details about how we actually extend the standard will be presented in Section 4.3.2 and Section 5.3.3.

3.2.1 DWARF Overview

Most modern programming languages are block structured: each entity is contained within another entity. For example, each file in a C program may contain multiple variable definitions and multiple functions. Within each C function, there may be several data definitions followed by executable statements. A statement may be a compound statement that in turn can contain data definitions and executable statements.

DWARF follows this model in that it is also block structured. Each descriptive entity in DWARF, except for the topmost entity which describes the source file, is contained within a parent entity and may contain children entities, which may represent types, variables, or functions for instance.

3.2.2 Debugging Information Entry

DWARF uses a series of basic descriptive entities called *Debugging Information Entry* (DIE) to define a low-level representation of a source program. A DIE has a *tag*, which specifies what the DIE describes and a list of *attributes*, which fill in details and further describes the entity. A DIE, except for the top-most, is contained in or owned by a parent DIE and may have children DIEs. Hence, a DIE, or a group of DIE together, provide a description of a corresponding entity in the source program, be it a function, a parameter, a variable or a label. Attributes may contain a variety of values: strings (such as function or variable name), constants (such as the address of a global variable), or references to another DIE (such as for the type of a variable).

DIEs can be split into two general types: those that describe data and those that describe executable code. The former will be used to describe variables referred in a property, while the latter will be use to represent the program point at which the property needs to be evaluated or verified.

3.2.2.1 Describing Data

Variable DIEs have a name which represents a chunk of memory (or register) that can contain some kind of values. This kind of values is described by the type DIE—which specifies the encoding (such as signed binary integer), and the size in bytes—associated to the variable. What distinguishes a variable is where its value is stored and its scope.

The scope of a variable defines where the variable is known within the program and is determined by where the variable is declared. This relation is naturally represented by declaring the variable DIE as a child of the DIE describing the scope within which the variable is declared.

Most variable DIEs have a location attribute that describes where the variable is stored. In the simplest of cases, a variable is stored in memory and has a fixed address, or more precisely, one that is a fixed offset from where the executable is loaded. But many variables, such as those declared within a C function, are dynamically allocated on the stack, and locating them requires adding a fixed offset to a

frame pointer. In other cases, the variable may be stored in a register. In either case, DWARF uses location expressions to describe how to locate the data represented by a variable. A location expression contains a sequence of operations (such as dereferencing, adding or subtracting) and values (such as a constant, or a register name) which details how to locate the data. DIEs describing variables whose values are constants and not represented by an object in the address space of the program do not have a location attribute but a constant value attribute instead. The value of this attribute is the actual constant value of the variable, represented as it would be on the target architecture. Debuggers use variable's location attribute to retrieve its value at a given execution point.

3.2.2.2 Describing Executable Code

We now present two types of DIEs describing the executable code that are of particular interest, as will be explained in Section 4.3.2 and Section 5.3.3.

Functions: DWARF describes functions with so-called subprogram DIEs. Such a DIE has a name, a source location triplet, and attributes that give the low and high memory addresses that the subprogram occupies (if it is contiguous), or a list of memory ranges (if the function does not occupy a contiguous set of memory addresses). The low PC address is assumed to be the entry point for the routine unless another one is explicitly specified. Additionally, the subprogram's return value is given by the type attribute (functions that do not return values do not have this attribute).

The subprogram DIE owns DIEs that further describe the function, such as variable DIEs representing parameters or variables defined inside the function.

Labels: A label is a way of identifying a specific source location. It is represented by a DIE which is owned by the DIE representing the scope within which the name of the label could be legally referenced in the source program. The label DIE generally has a name attribute and an attribute whose value is the address of the first executable instruction for the location identified by the label in the source program.

Now that we have described the LLVM compilation infrastructure and the DWARF debugging format, on top of which we will later implement our property-preserving compilation framework, let us present the security use-cases that highlight the needs for preserving source-level countermeasures during compilation, and that will be later used to illustrate our proposed solutions and validate their implementations (cf. Chapter 4 and Chapter 5).

3.3 Security Use-Cases

As stated previously, we would like to emphasize on the pivotal beneficial contribution of preserving properties through optimizing compilation flow in security engineering. Hence, as will be shown in the next chapters, our proposed solutions are validated and evaluated on a set of security countermeasures that would benefit from property preservation. These countermeasures are commonly introduced at the source level, but compilers may not understand their purpose and optimize them away. As a workaround, security engineers currently attempt to confuse the

compiler by resorting to compiler-dependent coding tricks. Nevertheless, this is obviously error-prone and not future-proof as compilers are getting better at spotting and removing "unnecessary" code (Simon, Chisnall, and Anderson, 2018). Furthermore, in some cases, notably for countermeasures against fault attacks, there exists no programming trick to reliably preserve source-level protections. As a last resort, the current approach adopted by the community is to compile the application with optimizations turned off.

In the following, we present each of these security use-cases using the same structure:

1. Presentation of the security threat as well as the source-level protection scheme;
2. Explicit statement of the security protection-derived properties that need to be preserved by the compiler, in order to conserve the protection itself;
3. Explanation of the security issue introduced by compiler optimizations;
4. Description of existing programming tricks (when available) to prevent compilers from altering the protection.

As a side note, we consider source programs written in C, since performance-critical applications and firmware of embedded systems, which are usually targets of physical attacks, are traditionally written in C. We use the informal semantics of the C standard (ISO/IEC 9899-2011) (ISO C11 Standard, 2011) as reference. As a simplifying assumption, we only consider deterministic, sequential C programs with well defined behavior, avoiding cases where the compiler may take advantage of undefined behavior to trigger optimizations. This assumption is consistent with widespread coding standards for secure code.

3.3.1 Sensitive Memory Data Erasure

Use-case description: First, we consider the security issue of sensitive data leakage, described in Section 1.1.2, which has been extensively studied in secure compilation, as illustrated in Section 2.2.3.2 and Section 2.2.3.3. This example is also usually used to demonstrate the correctness-security gap of optimizing compilers (Balakrishnan and Reps, 2010; D'Silva, Payer, and Song, 2015), as it is a major threat for applications. A security countermeasure consists in erasing a secret buffer allocated on the stack after usage. It will be illustrated in mbedTLS's implementation of *Rivest-Shamir-Adleman* (RSA) encryption and decryption (Bakker, 2019), called `erasure-rsa-enc` and `erasure-rsa-dec` in the following.

Note that this countermeasure deals with leakage from sensitive data in memory only. Leakage from data in registers or stack locations is not supported, because it incurs erasing any possible register or stack location where sensitive values may have resided in a function, which is not easily expressed as a source or IR-level property (Simon, Chisnall, and Anderson, 2018). This would also require an advanced data-flow analysis scheme to determine the list of registers and stack slots to be erased.

Security protection-derived property: As stated in Section 1.1.2, the countermeasure scheme can be equivalently expressed as a write of the zero value to the secret buffer, before returning from the function allocating the buffer. Therefore, the security protection-derived property stipulates that every element of the secret buffer holds the zero value before the function returns; we refer to this property as *secret erasure property* in the following.

Security issue due to compiler optimizations: Most compilers will spot that the buffer is not accessible after the function returns, removing the call to `memset()` as part of "dead store elimination".

Current solutions to preserve the protection: Existing work acknowledged this issue and surveyed the different techniques used to effectively erase memory data in C programs (Yang et al., 2017; Simon, Chisnall, and Anderson, 2018; Percival, 2014). The easiest way to ensure the security property is to replace the call to `memset()` by another function that guarantees that memory will be erased such as Microsoft `SecureZeroMemory` or C11 `memset_s`. Unfortunately, this technique relies on a platform-provided function which is not universally available. Worse, `memset_s` which is part of the optional Appendix K of the C standard (ISO/IEC 9899-2011) (ISO C11 Standard, 2011) has no mainstream support: it is not provided (yet) by common C standard libraries. As a result, developers have to come up with backup solutions.

Several techniques attempt to hide the semantics of the erasure operation from the compiler. The intuition behind this is, if the compiler does not recognize that an operation is erasing memory, it will not remove it. The simplest way to achieve this is to implement the erasure operation in a separate compilation unit. However, this is not reliable when *Link-Time Optimization* (LTO) is enabled, which can merge all the compilation units into one, giving the compiler a global view of the whole program. Compiler can then recognize the erasure operation and remove it. Another popular technique for hiding erasure operation from the compiler is to call the erasure function via a volatile function pointer. This has been adopted by popular cryptographic libraries such as OpenSSL (The OpenSSL Project, 2003) or mbedTLS (Bakker, 2019), as shown in Listing 3.1. However, while the C11 standard requires the compiler to read the value of volatile-qualified `memset_func` from memory, the compiler is not required to call `memset` if it can compute the same result by other means. For instance, a compiler may still load `memset_func` into a register, comparing it to `memset`, and perform the function call only if they differ.

```
typedef void *(*memset_t)(void *, int, size_t);
static volatile memset_t memset_func = &memset;
// Used in OpenSSL.
void OPENSSL_cleanse(void *ptr, size_t len) {
  memset_func(ptr, 0, len);
}
// Used in mbedTLS.
void mbedtls_zeroize(void *buf, size_t len) {
  memset_func(buf, 0, len);
}
```

LISTING 3.1: Erasure implementation using volatile function pointer.

There exists other techniques attempting to force the compiler to include the erasure operation without hiding its nature. One way to achieve this consists in using specially-crafted, complicated computation that obfuscates the code. Older version of OpenSSL, shown in Listing 3.2, is such an example. This implementation read and writes the global variable `cleanse_ctr` which provides garbage data to fill the memory to be erased. Since accesses to global variables have a global impact on the program, the compiler cannot determine that this function is useless without extensive interprocedural analysis, which is expensive. However, this is overly complicated and the generated code is less compact and slower. Furthermore, while this

appears effective in practice, there is no guarantee that compilers will not someday be aware of this trick and optimized it away.

```
unsigned char cleanse_ctr = 0;

void OPENSSL_cleanse(void *ptr, size_t len) {
  unsigned char *p = ptr;
  size_t loop = len, ctr = cleanse_ctr;
  while (loop--) {
    *(p++) = (unsigned char)ctr;
    ctr += (17 + ((size_t)p & 0xF));
  }
  p = memchr(ptr, (unsigned char)ctr, len);
  if (p)
    ctr += (63 + (size_t)p);
  cleanse_ctr = (unsigned char)ctr;
}
```

LISTING 3.2: Erasure implementation using code obfuscation.

Lastly, one may try to force the compiler to perform erasure by using a volatile-qualified data pointer. Older version of mbedTLS, shown in Listing 3.3, is one of the implementations based on this idea. The memory to be erased is written via a pointer-to-volatile `p` in the while loop. Similar to the obfuscation technique, this also leads to slower code, as it writes zeros one byte at a time. Furthermore, this behavior is implementation-specific, as it is actually not guaranteed by the C11 standard. In fact, only accesses to an object that has volatile-qualified type are required to be performed; accessing a non-volatile object via pointer-to-volatile may or may not be considered such an access. To make matters worse, there is also no guarantee that compilers will remain ignorant and not transform this idiom. For example, a discussion among LLVM developers concluded that it was valid to remove volatile stores if (1) the address is a stack allocation that does not escape and (2) the result of the store can be proven never to be read (Simon, Chisnall, and Anderson, 2018).

```
void mbedtls_zeroize(void *buf, size_t len) {
  volatile unsigned char *p = (volatile unsigned char *)buf;
  while (len--)
    *p++ = 0;
}
```

LISTING 3.3: Erasure implementation using volatile data pointer.

3.3.2 Masking Computation Order

Use-case description: Masking is one of the most widespread countermeasures against power analysis attacks on cryptographic algorithms (Ishai, Sahai, and Wagner, 2003; Rivain and Prouff, 2010). The idea is to combine the secret value with random noise, then to perform intermediate computations using this masked value, and to unmask the result at the end, so that all computations are statistically independent of the original secret value. In order to further maintain this statistical independence from the secret value, notably when only a limited set of masks is available, mask swapping can be necessary at some point (Herbst, Oswald, and Mangard, 2006). This mask swapping scheme is illustrated in Listing 3.4. The secret `k` is first masked with `m` (line 1), then is re-masked with `mpt`, so that finally previous mask `m` can safely be removed from `k` (line 3). This countermeasure will be illustrated in a masked implementation of *Advanced Encryption Standard* (AES) (Herbst, Oswald, and Mangard, 2006), named `mask-aes` in the following.

```
1 k ^= m;
2 ... // k, masked with m, can be safely used here
3 k = (k ^ mpt) ^ m;
4 ... // k, masked with mpt, can be safely used from here
```

LISTING 3.4: An example of mask swapping scheme.

Security protection-derived property: The effectiveness of this countermeasures lies in the specific computation order: re-masking must take place before the removal of the previous mask to avoid exposing the unmasked secret key value. In other words, the protection-derived property requires that the computation order of "re-masking then unmasking" is respected; we refer to this property as *instruction ordering property* in the following.

Security issue due to compiler optimizations: Although the instruction ordering property holds at the source level, it has been reported that compiler optimizations can produce an semantically equivalent, yet unsafe program by swapping the `^` operations such that the unmasking operation takes place before re-masking, i.e., turning line 3 into `k = k^(mpt^m)`, thus making the transformed program vulnerable to power analysis attacks (Bayrak et al., 2013; Eldib and Wang, 2014). This happens because of the `^` operation's associativity and that compilers have perfectly the right to reorder associative operations, as programs before and after this transformation produce matching observable behaviors, with respect to the C standard. Note how the programmer has intentionally put the parentheses to express the security property saying `k` has to be re-masked with `mpt` before removing the previous mask `m`.

This example highlights the challenges for respecting the computation order of associative operations, as explicitly written in the source code, with optimizing compiler. The C programming language is defined in terms of an abstract machine producing observable behaviors. A compiler can optimize a conforming program as long as the generated observable behaviors match the one from the C language abstract machine; it is thus free to reorder associative operations, even with proper parenthesizing. To make things worse, this is independent of the optimization level, and programmers usually have no control over it.

Current solutions to preserve the protection: A mitigation trick is shown in Listing 3.5. One can first declare a volatile-qualified temporary variable `tmp` holding the result of the re-masking operation (line 3), then later use it for unmasking (line 5). Furthermore, to ensure that the result of re-masking has been stored into `tmp` before it is loaded for unmasking, one can insert a memory barrier implemented by inline assembly statement (line 4). Again, similar to some erasure techniques presented in Section 3.3.1, this trick is implementation-specific, not future-proof and might also result in slower code, as it prevents the compiler from caching values in registers, and furthermore prevent optimizations from moving all memory accesses across the introduced barrier.

```
1 k ^= m;
2 ... // k, masked with m, can be safely used here
3 volatile uint8_t tmp = k ^ mpt;
4 __asm__ __volatile__("" : : : "memory");
5 k = tmp ^ m;
6 ... // k, masked with mpt, can be safely used from here
```

LISTING 3.5: An example of mask swapping scheme.

3.3.3 Step Counter Incrementation

Use-case description: The next countermeasure is designed to enhance the resilience against fault attacks that induce unexpected jumps to any location in the program (Lalande, Heydemann, and Berthomé, 2014; Heydemann, Lalande, and Berthomé, 2019). More specifically, this source-level scheme aims at detecting harmful attacks that disrupt the control flow by not executing at least two adjacent statements of C programs. It deals with different high-level control flow constructs, namely straight-line flow of statements, function calls, conditional constructs (if-then, if-then-else or switch) and loop constructs. We refer to this technique as *Step Counter Incrementation* (SCI); it may be seen as a very fine-grained form of *Control Flow Integrity* (CFI). This countermeasure will be illustrated in two well-known smart-card benchmarks: PIN authentication (Dureuil et al., 2016) and AES encryption (Levin, 2007), called `sci-pin` and `sci-aes`, respectively, in the following.

To secure the control flow integrity of a whole block of straight-line statements, the latter is associated with a dedicated counter which is first initialized outside of the block, then incremented after each C statement inside the block from the original program. At the exit of the block, a check of the expected value of the counter is performed and if this fails, the program execution stops.

To secure the control flow integrity of a conditional construct, two counters (for each branch of the construct) and one extra variable holding the condition value of the conditional flow are required. Their declarations and initializations are performed outside the construct. Both counters are checked, according to the condition value, at the exit of the conditional construct. Consider an example of this technique, with the protection code automatically inserted as C macros using the script from the original work, shown in Listing 3.6.

```
1  ...
2  unsigned short cnt_then = 0, cnt_else = 0;
3  if (cond) {
4    if (cnt_then != 0) killcard(); // CHECK_INCR expanded
5    cnt_then++;
6    a = b + c;
7    if (cnt_then != 1) killcard(); // CHECK_INCR expanded
8    cnt_then++;
9  }
10 if (!((cnt_then == 2 && cnt_else == 0 && cond) || // CHECK_END_IF_ELSE
11       (cnt_then == 0 && cnt_else == 0 && !cond))) // expanded
12   killcard();
```

LISTING 3.6: An example of SCI scheme.

The protection first defines two step counters `cnt_then` and `cnt_else` associated to each branch of the conditional construct (line 2). It then steps `cnt_then` after every C statement of the branch "then", including the test statement (lines 5 and 8). Before any incrementation, a check of the expected value of `cnt_then` is performed (lines 4, and 7). The branch "else" is actually empty, `cnt_else` is not incremented at all. `cnt_then` and `cnt_else` are finally checked against their expected values at the exit of the conditional construct, according to the value of the condition `cond` (lines 10 and 11). Whenever an inserted check fails, a fault is detected and a handler `killcard()` is called (lines 4, 7 and 12).

Similarly, to secure the control flow integrity of a loop construct, a counter and an extra variable holding the value of the loop condition are required. The latter is needed at the end of the loop to verify the correct execution of the loop body and the correct termination of the loop.

Security protection-derived property: In order for the protection to be effective, two following protection-derived properties are crucial: (1) counters need to be correctly incremented and checked against their expected values; and (2) every incrementation needs to be performed *after* the execution of the corresponding source statement. Note that the latter also implies the former: incrementations have to be preserved in the first place in order to be correctly scheduled later. In other words, the proper interleaving of functional and protection code needs to be preserved; we refer to this property as *code interleaving property* in the following.

Security issue due to compiler optimizations: Clearly, the countermeasure is fragile against compiler optimizations which do not understand the intention of generating secure code. As long as faults are not modeled in compilers, counter checks can be removed—their conditions are trivially true in a "fault-free" semantics of the program—whereas counter incrementations can be grouped into a single addition statement or even removed.

Current solutions to preserve the protection: Unfortunately, there currently exists no reliable workaround preventing optimizations from violating these security properties. As a result, practitioners making use of this source-level hardening scheme have to disable compiler optimizations to guarantee the protection's effectiveness.

3.3.4 Control and Data Flow Redundancy

Use-case description: The next countermeasure that we consider is the loop protection scheme already described in Section 2.2.4.3, which aims at ensuring the security property stating that a sensitive loop must always perform the expected number of iterations and take the right exit, even in presence of fault injections. The original loop hardening algorithm (Proy et al., 2017) operates on the LLVM IR and the authors has carefully investigated and analyzed different compilation passes in order to select a beneficial position for the loop hardening pass in the compilation flow, with respect to downstream and upstream passes, so that the countermeasure is preserved in the generated binary. We argue that, if we are able to reliably maintain the protection code during compilation, such an investigation is not mandatory and can thus be dismissed. Indeed, we attempt to make the approach more generic by implementing it at source level. We implemented the loop hardening scheme on the core loop of PIN authentication (a `memcmp`-like function) (Dureuil et al., 2016), named `loop-pin` in the following.

```
1 int memcmp(char *a1, char *a2, unsigned n) {
2   for (unsigned i = 0; i < n; ++i) {
3     if (a1[i] != a2[i]) {
4       return -1;
5     }
6   }
7   return 0;
8 }
```

LISTING 3.7: Original `memcmp()` implementation.

More specifically, given an implementation of `memcmp()` loop shown in Listing 3.7, we apply the countermeasure scheme to this function and obtain `secure_memcmp()` in Listing 3.8. We duplicate the loop counter `i` (as well as all computations involving it,

lines 2 and 3) and loop-independent variables being used in the loop body (`n` in this case, line 2). Furthermore, we insert redundant computations of the exit condition at every iteration of the loop (line 4), as well as at the loop exit (lines 7 and 14). We also verify that the values of the duplicated loop-independent variables at every loop exit are correct with respect to the values of their original counterparts (lines 9 and 16).

```
1  int memcmp(char *a1, char *a2, unsigned n) {
2    unsigned i, i_dup, n_dup = n;
3    for (i = 0, i_dup = 0; i < n; ++i, ++i_dup) {
4      if (i_dup >= n)
5        fault_handler();
6      if (a1[i] != a2[i]) {
7        if (a1[i_dup] == a2[i_dup])
8          fault_handler();
9        if (n_dup != n)
10         fault_handler();
11       return -1;
12     }
13   }
14   if (i_dup < n)
15     fault_handler();
16   if (n_dup != n)
17     fault_handler();
18   return 0;
19 }
```

LISTING 3.8: Securing `memcmp()` loop.

Security protection-derived property: In order to ensure the loop iteration number integrity and guarantee that the right exit is always taken, the property that needs to be preserved behind this protection scheme consists in the preservation of duplicated variables `i_dup` and `n_dup`, as well as the computations and checks involving these duplicates; we refer to this property as *redundancy preservation property* in the following.

Security issue due to compiler optimizations: Since the approach relies on redundant computations, the difficulty lies in preserving the protection from optimizations: redundant operations are ideal candidates to be optimized away by compilers. Indeed, as detailed in Section 2.2.4.1 and Section 2.2.4.3, redundancy is eliminated in various places in the compilation flow, from optional optimization passes such as "instruction combining" or "dead code elimination"—which can be avoided by disabling optimizations—to mandatory transformation passes such as "instruction selection" which lowers IR instructions into machine specific representation and thus cannot be bypassed.

Current solutions to preserve the protection: Although redundancy is commonly used to implement countermeasures against fault injection attacks (Witteman, 2018), there exists, perhaps surprisingly, no well grounded approach to preserve it through optimizing compilation, other than introducing redundant instructions at the very end of the compilation pipeline, as illustrated in Section 2.2.4.2. Unfortunately, this is not always desirable, or even feasible, as some countermeasures require high-level information about the program such as the CFG or loop structures. One may attempt to preserve redundant variables by using the `volatile` keyword, but as explained in Section 3.3.1 and Section 3.3.2, this technique is not future-proof and leads to slower code by introducing extra loads and stores to the program. Furthermore,

this is particularly undesirable for applications submitted to fault injections: it also produces larger code and thus increases the overall attack surface, which in turn increases the probability for an attacker to find and exploit vulnerabilities (Bréjon et al., 2019).

3.3.5 Constant-Time Selection

Use-case description: Another well-known, yet hard to achieve example of security property is selecting between different values, based on a secret selection variable, in constant time. As shown in Section 2.2.3.1, this means the generated code for the selection operation must not contain any jump or memory access pattern conditioned by the secret selection variable, otherwise the execution time of the operation will depend on which value is selected, thus leaking the secret selection variable. Cryptography libraries resort to data-flow encoding of control flow, bitwise arithmetic at source level to avoid conditional branches, but this fragile constant-time encoding may be altered by an optimizing compiler. This security property will be illustrated in mbedTLS's RSA decryption (Bakker, 2019) and a self-written RSA exponentiation using Montgomery ladder (Simon, Chisnall, and Anderson, 2018), respectively called `ct-rsa` and `ct-montgomery` in the following.

```
uint32_t ct_select_vals_1(uint32_t x, uint32_t y, bool b) {
  signed m = 0 - b;
  return (x & m) | (y & ~m);
}
```

(a) Constant-time selection between two values, version 1

```
uint32_t ct_select_vals_2(uint32_t x, uint32_t y, bool b) {
  signed m = 1 - b;
  return (x * b) | (y * m);
}
```

(b) Constant-time selection between two values, version 2

```
uint64_t ct_select_lookup(const uint64_t tab[8], const size_t idx) {
  uint64_t res = 0;
  for (size_t i = 0; i < 8; ++i) {
    const bool cond = (i == idx);
    const uint64_t m = (-(int64_t)cond);
    res |= tab[i] & m;
  }
  return res;
}
```

(c) Constant-time selection from lookup table

Listing 3.9: Constant-time selection attempts.

For example, consider different functions from Listing 3.9. Listing 3.9a and Listing 3.9b represent two implementations of constant-time selection between two values `x` and `y` based on a secret selection bit `b`, while Listing 3.9c shows an example where the programmer wishes to select a value from the lookup table `tab` while hiding the secret lookup index `idx`. All these functions are carefully designed to contain no branch conditioned by the secret value: a bitmask `m` is created from the secret value using arithmetic tricks, then is in turn used to select the wanted values.

Security protection-derived property: One can equivalently express the constant-timeness as the preservation of the arithmetic idioms, which are guaranteed, to be

transformed into machine code with the same constant-timeness as the source program; we refer to this property as *constant-time selection property* in the following.

Security issue due to compiler optimizations: It has been reported that these arithmetic idioms are not preserved by LLVM, thus the code generated is not guaranteed to be constant-time. For instance, the compiler introduces a conditional jump based on the secret value when compiling, with optimizations enabled, the first two functions for IA-32 (Simon, Chisnall, and Anderson, 2018), or the last function for both IA-32 and x86-64 (LLVMdev, 2020). In fact, the compiler recognizes these arithmetic idioms and transforms them into the ordinary selection operations, which in turn can be lowered into conditional jumps, depending on the target architecture.

Current solutions to preserve the protection: The community is desperately in search of a reliable way to prevent the compiler from spotting and optimizing the constant-time idioms. As presented in Section 2.2.3.1, other workarounds introduce to the compiler a specially-crafted operation that will be ultimately compiled into a conditional move instruction (if available in the target architecture) (Simon, Chisnall, and Anderson, 2018; Barthe et al., 2019). However, this solution is target-dependent and lack of generalizability: it only supports the operation of selecting between two values, and needs to be modified in order to implement the constant-time selection from lookup table from Listing 3.9c for instance.

3.4 Discussion

In this chapter, we presented the LLVM compiler toolchain, on top of which we will implement our property-preserving compilation framework. We detailed its overall three-phase architecture as well as its key features needed for our implementations. We also introduced the DWARF debug data format, which will be used to represent, at binary level, the properties propagated through the compilation process. We finally described our selected security use-cases including different protection schemes, which will be used for validation and evaluation of our approaches.

Having introduced the different basic blocks needed to explain our approaches, in the next two chapters, we will detail our proposed solutions to preserve program properties through optimizing compilation, their implementations in LLVM, how we represent the propagated properties in the executable binary using the DWARF format, and the validation of our two solutions on a reference suite of functional properties, and notably on our selected set of security use-cases.

Chapter 4

Automated Property Preservation at Compile-Time

The goal of this thesis is to devise a mechanism preserving, along an optimizing compilation of C code to machine code, both the semantics of functional properties—which are external to the program itself, and the security countermeasures introduced in the source program to guarantee some security properties. Section 3.3 discusses some scenarios featuring different security countermeasure schemes, and for each of these schemes, states the security protection-derived property which, if preserved during compilation, implies that the associated countermeasure will also be preserved by the compiler. As will be shown in this chapter, these protection-derived properties can be equivalently encoded as a special type of functional properties, so that we can apply the same mechanism preserving functional properties to preserve the protection-derived properties.

However, preserving external functional properties in an optimizing compilation flow is hard. This boils down to (1) propagating the properties through the compilation flow in parallel with program transformations, (2) maintaining their consistency with the code undergoing transformations and (3) embedding the properties into the compiled binary without interfering with the executable code itself. Furthermore, as illustrated through various security scenarios presented in Section 3.3, we also need to prevent optimizations from invalidating the properties by, for instance, removing referred variables or reassociating expressions. In other words, we need a mechanism to maintain their presence in the code undergoing transformations. In this chapter, we present our first proposed approach to address this problem, automatically deployed at compile-time and relying on I/O side-effects. As a result, we refer to this solution as *I/O-barrier-based* approach.

In this chapter, We first start by defining the notion of functional property and what it actually means for the compiler. We also introduce the concept of observation trace and how it is used to define the preservation of functional properties in a compilation flow.

Given these definitions, we next describe our compilation approach to preserve functional properties. This involves (1) capturing and translating source-level properties through lowering passes and intermediate representations, such that data and control flow optimizations will preserve their consistency with the transformed program, and (2) carrying properties as debug information down to machine code. It is worth noting that instead of trying to *transform* the properties in concert with program transformations, for example by devising property transformation rules for each compiler transformation—which is not feasible for a modern optimizing compiler including hundreds of optimizations—we propose an alternative approach that introduces additional constraints that every transformation needs to respect. These

constraints ensure that all source denotations referred in program properties are preserved (i.e. not optimized out), thus maintaining the semantics of these properties. Note that the constraints might prevent some optimizations from happening. For example, while a property verifying the value of the loop count at each iteration does not prevent "loop unrolling" from being performed, it will block transformation such as "loop reversing" from turning an original counter-incremented loop into a counter-decremented one. However, this is a performance-generalizability trade-off that we deliberately accept, in order to devise an optimization-agnostic approach for preserving program properties that can be implemented in an aggressively-optimizing compiler used in industrial settings.

To validate the soundness and efficiency of the approach, we implement the proposed mechanism in LLVM and conduct experiments considering a reference suite of functional properties as well as established security use-cases presented in Section 3.3. Furthermore, we also illustrate the idea of automatically providing binary analysis with additional properties by plugging our compilation framework into a robustness analysis tool, thus automating the whole analysis process.

4.1 Definitions

Let us start by laying the foundation of our approach, introducing some required definitions. In the following, the program may refer to any representation used by different steps of the compilation process: source, IR or machine code level.

Definition 4.1.1 (Program state)**.** A program state is defined by

- a distinguished value of the program counter π denoting a program point;
- a finite set V of $(\text{variable}, \text{value})$ pairs for all program-defined variables at π;
- a large but finite set M of $(\text{memory\_location}, \text{value})$ pairs stored in main memory at π.

Definition 4.1.2 (Program partial state)**.** A program partial state is a subset of a (complete) program state. It is defined by a program counter π, a set $V' \subseteq V$ and a set $M' \subseteq M$. A program partial state holds the $(\text{variable}, \text{value})$ pairs in V' and the $(\text{memory\_location}, \text{value})$ pairs in M', both at π.

Note that Definition 4.1.1 (Program state) and Definition 4.1.2 (Program partial state) refer to a generic notion of program variable, either a source-level denotation, an SSA value, or a register in a low-level representation. As will be presented in Section 4.3, we do not take into account the precise nature of the variable but maintain a mapping from denotations referred in source-level functional properties to their corresponding representations at any given compilation step.

Definition 4.1.3 (Functional property)**.** A *functional property* specifies the program behavior at a given program point by exclusively referencing its variables and memory locations. It takes the form of a pair $(\mathit{Formula}, \mathit{ObsPt})$, where *Formula* is a propositional logic formula expressing the behavioral properties of the program; *ObsPt* denotes the program point called *observation point* at which property *Formula* is expected to hold.

Given a functional property $(\mathit{Formula}, \mathit{ObsPt})$, we call *ObsVar* and *ObsMem* the sets of all *observed variables* and *observed memory locations* occurring in *Formula*. The

functional property defines a *partial state* containing the (variable, value) pairs and (memory_location, value) pairs of all observed variables and observed memory locations at observation point *ObsPt*.

```
1 k ^= m;
2 ...
3 mpt = get_val();
4 tmp = k ^ mpt;
5 /* Property: tmp == k ^ mpt */
6 k = tmp ^ m;
7 ...
```

LISTING 4.1: An example of functional property for a mask swapping scheme.

As an example of functional property, we consider the mask swapping scheme from Section 3.3.2 again. The code shown in Listing 4.1 has been rewritten for illustration purpose. Let us remind that the security property stipulates that all variables must be statistically independent of the secret. Therefore, variables that depend on secret must be masked throughout the whole program execution, in every computation involving those. The protection scheme implements a mask swapping operation, and its effectiveness requires a protection-derived property ensuring that re-masking must take place before the removal of the previous mask to avoid exposing the unmasked secret key value. Thereby, the protection-derived property can be expressed with the boolean expression `tmp == k ^ mpt` at a specific point (line 5), and that this value of `tmp` is indeed used to compute the new value of `k` (line 6). In contrast to the complete program state that includes pairs for variables `k`, `tmp`, `mpt`, and `m`, the partial state defined by the considered functional property `tmp == k ^ mpt` only contains pairs for `k`, `tmp` and `mpt`.

Intuitively, a functional property is preserved by a transformation when there exists, in the transformed program, a program point where all variables from the partial state defined by the property hold their expected values. As a result, functional property must *act as a barrier at ObsPt for all memory accesses to locations in ObsMem and definitions of variables in ObsVar*: no definition of a variable and no access to a memory location observed by *Formula* may be moved across *ObsPt* through program transformations.

It is worth noting that Definition 4.1.3 (Functional property) can be generalized to invariant properties of a control flow region: the latter may be considered as a set of functional properties $(Formula, ObsPt_i)$ for all program points $ObsPt_i$ belonging to the region.

Definition 4.1.4 (Observational property)**.** An *observational property* can be defined as a special functional property which does not specify any program behavior but a list of observed entities and an observation point *ObsPt*. The property thus defines a partial state containing the (variable, value) pairs and (memory_location, value) pairs of all these observed entities at *ObsPt*.

In short, observational properties specify to the compiler a list of values that will be externally observed, and thus impose the preservation of these values on program transformations. Observational properties are primarily used to encode security protection-derived property, which, if preserved, allows for preservation of the associated security protection. For instance, consider the example from Listing 4.1. In order to express the requirement of "re-masking before unmasking", needed

for the effectiveness of the masking scheme, one only needs to ensure that `tmp`—which holds the result of re-masking operation—is actually available at the observation point, *and is indeed used in the subsequent unmasking operation*. In other words, instead of verifying the complete boolean expression `tmp == k ^ mpt` at the observation point, it is sufficient to *observe* `tmp` to make sure its value is available at this point. We also need to ensure that the observed value of `tmp` is actually used for unmasking. More details on the encoding of security protection-derived property as observational property, as well as its application to security countermeasure preservation will be discussed in Section 4.4.3.

Definition 4.1.5 (Observation trace)**.** Given a finite set of functional properties FP, an *observation trace* is a finite sequence of program partial states defined by the properties in FP. Again, we restrict ourselves to sequential, deterministic programs.

Definition 4.1.6 (Functional property preservation)**.** Given a program P and a transformation τ that applies to P producing a semantically equivalent program P'—i.e. with the same I/O behavior—τ is said to *preserve all functional properties* in P if the observation traces produced by P and P', given the same input, are equal—i.e. they have the same sequence of partial states.

This is a rather conservative definition as it does not allow any reordering of partial states. We will relax it in the solution described in Chapter 5, but consider this simpler definition for now, so that we can focus on explaining the principle of our approach, before inspecting a more optimized solution.

Note that Definition 4.1.6 (Functional property preservation) does *not* define a notion of *program point preservation*. Only pairs $(Formula, ObsPt)$ are preserved, as defined by the associated partial states in the observation traces of the original and transformed programs. Still, a program transformation τ maps functional properties $(Formula_i, ObsPt_i)$ in a program P into functional properties $(Formula'_j, ObsPt'_j)$ in a program P'. One logic formula $Formula_i$ in P may correspond to one or more formulas in P' (for instance, when $ObsPt_i$ is inside a loop and transformation τ is loop unrolling, which would duplicate the loop body) up to the renaming of its variables and the rearrangement of its memory locations, as long as evaluating the formula at observation point $ObsPt'_j$ results in the same value at the same relative position in the trace.

4.2 An Approach for Preserving Functional Properties

As explained previously, optimizations are free to remove variables, reorder or remove instructions as long as they do not change the program's observable behavior (cf. Chapter 1 and Section 2.2.4). Preserving functional properties according to Definition 4.1.6 (Functional property preservation) implies (1) preventing the compiler from removing any observed variable or memory location, (2) maintaining the correspondence between the observed variables or memory locations and their values at the associated observation points, and (3) blocking the movement of accesses to observed memory locations and definitions of observed variables across the functional property observing them (the barrier effect of functional properties). This section describes our solution.

Production compilers such as gcc or LLVM generally have an IR in SSA form, and we first describe our solution tailored for the SSA properties.

The SSA form ensures that any program point has a unique reaching definition for live SSA variables. As a result, a source variable v is represented by multiple SSA

variables $V = \{v_1, \ldots, v_n\}$. Note that multiple SSA variables corresponding to the same source variable may be alive at a given program point.

A natural approach to propagate program functional properties consists in tracking reaching definitions of all observed variables across IRs. This would imply finding, after each program transformation, an observation point at which all reaching definitions of the observed variables exist. However, the existence of such program point is not always guaranteed by compiler optimizations, or worse, a given reaching definition might be optimized away. Hence we propose an approach that propagates *by construction* both the observation point and the reaching definitions of the observed variables defined in the source program or any IR.

Given a functional property, we need to ensure the correct values of the observed variables and the values stored at the observed memory locations at the associated observation point. For memory locations, one may enforce these constraints by inserting a compiler fence, i.e. an instruction with a memory side-effect, see e.g. Listing 3.5. Enforcing the same constraints for the observed variables implies preserving the reaching definitions of every observed variable at the observation point: we have to make sure that these definitions exist and take place before the observation point. This is generally not guaranteed by optimizing compilers since optimizations such as code motion do not know about such constraint on the presence of functional property.

```
%k2 = xor %k1, %m
...
%mpt = call get_val()
%tmp = xor %k2, %mpt
// Prop: %tmp == %k2 ^ %mpt
%k3 = xor %tmp, %m
...
```

(a) Functional property preserved

```
%k2 = xor %k1, %m
...
%mpt = call get_val()
%tmp = xor %k2, %m
// Prop: %tmp == %k2 ^ %mpt
%k3 = xor %tmp, %mpt
...
```

(b) Functional property invalidated

Listing 4.2: An example of functional property for a mask swapping scheme in SSA.

As an example, consider the property `tmp == k ^ mpt` from Listing 4.1. We illustrate the concern in Listing 4.2, which shows the same code example in SSA form: the source variable `k` is translated into 3 different SSA variables `%k1`, `%k2` and `%k3`, where `%k2` is the reaching definition of `k` for the observation point of the property. Similarly, variable `mpt` from the source program is represented by `%mpt`, while `tmp` is now `%tmp` . Variables in the property have also been renamed accordingly into `%tmp`, `%k2` and `%mpt`. The snippet in Listing 4.2a represents the program with the correct computation order with regards to the considered functional property, shown in green, while one in Listing 4.2b shows a semantically-equivalent program but with the functional property being invalidated, shown in red. Indeed, because the C standard does not define a specific evaluation order for associative operators, compilers can arbitrarily transform the original program into either SSA code, and this is done regardless of whether optimizations are enabled (ISO C11 Standard, 2011).

The key idea of our solution is to tie together the reaching definitions of observed variables and the observation point. We achieve this by inserting different instructions to the program.

Given a functional property, we first materialize the associated observation point using an *opaque, I/O side-effecting compiler fence*. This guarantees that all memory accesses—including ones to the observed memory locations—cannot be moved across

the observation point, which in turn ensures the correct values stored at these observed memory locations at the observation point. In other words, the compiler fence implements the barrier effect of functional properties for observed memory locations. Furthermore, the compiler fence is opaque and I/O side-effecting, which means that, not only it cannot be removed by compiler optimizations, but the relative order of different observation points (and thus functional properties) are guaranteed to be preserved.

Next, we need to implement the barrier effect of functional properties for observed variables. To this end, for each such variable, we introduce an *opaque, I/O side-effecting identity function* that takes the reaching definition of the variable as a parameter, and produces another value. The function is opaque, meaning that the compiler cannot analyze it and does not know about the identity relation between its input parameter and its output result; the output value is hence statically unknown to the compiler. We also replace the original value of the reaching definition, for all of its uses in the subsequent code, by the opaque value produced by the identity function. We thus call these identity functions *artificial definitions*. The artificial definitions are also I/O side-effecting, so that their relative ordering with respect to the compiler fence cannot be altered by the compiler. As a result, program transformations cannot move the artificial definition of any observed variable across the compiler fence. It is thus guaranteed that the reaching definitions of observed variables will always be available at a given observation point.

Algorithm 1. Artificial definitions in SSA

input: *Properties*, list of functional properties

```
1 for (Formula, ObsPt) ∈ Properties do
2     ObsVar ← variables of Formula;
3     CF ← compiler fence associated with ObsPt;
4     for OV ∈ ObsVar do
5         RD ← reaching definition of OV for ObsPt;
6         insertArtificialDefinition(RD, RD, After);
```

For every element in the set *ObsVar* of observed variables of a given functional property, Algorithm 1 is in charge of inserting artificial definitions into the program. Function $insertArtificialDefinition(D_v, Inst, \text{Before}|\text{After})$ creates an artificial definition producing an SSA variable v' from an SSA variable v defined by D_v, inserts it before (respectively after) instruction *Inst* and replaces all uses of v subsequent to the insertion point by v'. Notice that Algorithm 1 inserts artificial definitions right after the corresponding reaching definitions, which allows for maintaining the SSA form of the program.

Applying the algorithm to Listing 4.3a yields the code in Listing 4.3b : all observed variables in the property (`%k2`, `%tmp` and `%mpt`) have been renamed (respectively `%k2'`, `%tmp'` and `%mpt'`). Artificial definitions opaquely define new values for these observed variables. The compiler cannot make assumptions about these values and is forced to assign concrete locations to hold them. As a result, artificial definitions do prevent the elimination of the observed variables at the observation point (they may still be optimized in the rest of the code). It is worth noting that replacing subsequent uses of the original values of the observed variables (e.g. `%tmp`) by the corresponding opaque values (e.g. `%tmp'`) is crucial here, as this ensures that the unmasking operation actually uses the opaque value `%tmp'` defined by the artificial definition, and thus indeed occurs *after* the re-masking operation, needed for

```
%k2 = xor %k1, %m

...
%mpt = call get_val()

%tmp = xor %k2, %mpt

// Prop: %tmp == %k2 ^ %mpt
%k3 = xor %tmp, %m
...
```

(a) Original

```
%k2 = xor %k1, %m
%k2' = call artificial_def(%k2)
...
%mpt = call get_val()
%mpt' = call artificial_def(%mpt)
%tmp = xor %k2', %mpt'
%tmp' = call artificial_def(%tmp)
call obs_pt("%tmp' == %k2' ^ %mpt'")
%k3 = xor %tmp', %m
...
```

(b) Solution

Listing 4.3: Preserving functional property in SSA.

producing `%tmp'` itself.

4.3 Putting it to Work

In this section, we first describe our proposed representation for functional properties in different program representation levels from source code, through compiler IR, down to binary code, then detail how we implemented our mechanism for functional properties preservation in an optimizing compiler. As stated in Section 3.4, we consider source programs written in C, and choose to base our framework on the LLVM compilation infrastructure.

4.3.1 Functional Properties in Source Code

We capture functional properties as source program annotations, and use boolean expressions following C language syntax to represent functional properties. Our annotations are written directly in C source files using the following GNU C attribute syntax: `l: __attribute__((annotate("str")));`. The label `l` denotes the observation point and the functional property is expressed by the boolean expression `str`. Note that we do not restrict ourselves to this specific syntax, we believe that our solution could be adapted to other annotation mechanisms such as pragmas, with only minor modification.

4.3.2 Functional Properties in Machine Code

When attaching functional properties to machine code, one needs to make sure binary analysis tools are capable of extracting the corresponding representation of the observed variables and memory locations.

Debug information is generated by the compiler for the purpose of communicating source location, type and variable information to the debugger. Therefore, it already provides machine code with a detailed description of source-level variables as well as of memory locations. As mentioned in Section 3.2, it is thus very reasonable to extend the debug information to represent functional properties in the binary: the formulas expressing the functional properties are propagated and emitted into the debug section of the binary, while the binary representations of observation points and observed variables and memory locations are already provided by the standard debug information.

To this end, we introduce to DWARF—a widely-used and easily-extensible debug data format—new tags and attributes to represent source-level functional properties. A property `obs_pt: __attribute__((annotate("tmp == k ^ mpt")));` from the source program is represented by a DIE which contains:

- the formula string `"tmp == k ^ mpt";`
- a reference to the label DIE representing `obs_pt` which contains the machine code address corresponding to the observation point (cf. Section 3.2.2.2);
- references to the DIEs representing the observed variables and memory locations, in this case `tmp`, `k` and `mpt`. Each of these DIEs contains the location attribute allowing binary analysis tools to retrieve the values of these variables (cf. Section 3.2.2.1)

The property DIE is owned by the subprogram DIE representing the function containing the observation point at which the property occurs.

Unfortunately, it is common knowledge that debug information is a second-class citizen of compiler validation, and may not be accurate, or even not available in the presence of aggressive optimizations. For example, when a redundant calculation is optimized out, preserving the information about its result (such as variable locations) is not possible. A fortunate side-effect of inserting artificial definitions is to prevent most optimization passes from harming the observed variables' debug information, as artificial definitions actually preserves the values of such variables. However, we still had to fix a few critical remaining bugs in LLVM code computing and propagating the debug information, and filed bug reports for others (cf. Section 4.4.3). As an immediate benefit of leveraging (accurate) debug information, compiler passes do not have to worry about renaming variables observed in functional properties: debug information takes care of tracking the mapping of source to IR variables, down to machine code registers and stack locations.

4.3.3 Observed Variables: Multiple Definitions and Debug Information

Let us now consider the scenario where there exists multiple definitions for a given observed variable. As it stands today, debug information can only provide a *single* value for a *source* variable at a given line of code. When multiple live ranges corresponding to the same source variable overlap at a given code region, only the most recent one is reported in the variable location corresponding to the overlapping code region, stored in the DIE describing the variable in question. In the same spirit, when multiple live ranges get permuted, for a given program point, debug information only refers to the last definition, irrespectively of the initial program order. Such behavior may be observed after variable renaming and live range splitting, followed by code motion. This simplification is consistent with the common usage of debug information in debuggers, but it conflicts with our application to functional property preservation. This is the reason why we have to forbid any transformation from reordering definitions of *different occurrences of the same source variable*, as soon as one of these occurrences is observed by a functional property.

To illustrate this issue, let us consider the mask swapping scheme from Section 3.3.2 again. We expand the snippet of code implementing the re-masking operation with different assignments to `mpt` (lines 3, 5 and 10 from the source code snippet in Listing 4.4a). As illustrated in the SSA code snippet in Listing 4.4b, two SSA variables `%mpt1` and `%mpt2` stemming from the same source variable `mpt` and defined before the functional property will have to remain there (before the observation point);

```
k ^= m;
...
mpt = get_val1();
...
mpt = get_val2();
tmp = k ^ mpt;
// Prop: tmp == k ^ mpt
k = tmp ^ m;
...
mpt = 42;
...
```

(a) Source program

```
%k2 = xor %k1, %m
...
%mpt1 = call get_val1()
...
%mpt2 = call get_val2()
%tmp = xor %k2, %mpt2
// Prop: %tmp == %k2 ^ %mpt2
%k3 = xor %tmp, %m
...
%mpt3 = 42
...
```

(b) IR program

Listing 4.4: Example of multiple definitions of an observed variable for a mask swapping scheme.

conversely, SSA variables like `%mpt3` corresponding to later live ranges of `mpt` can only be defined after the functional property. As a sufficient condition to enforce this additional constraint, as soon as one variable is observed by the property, Algorithm 1 needs to be adjusted to actually *preserve the relative order* of "sibling" variable definitions stemming from the same source variable[1]. This may sound as overkill but it is currently necessary to prevent non-reaching definitions from clobbering the variable observed by the functional property. On the bright side, one advantage of this solution is that *functional properties do not need to be transformed to rename the observed variables* along the compilation flow; this removes the burden of modifying many compilation passes and also helps the interpretation of program binary analysis.

Algorithm 2. Artificial definitions considering debug information

input: *Properties*, list of functional properties

```
1  for (Formula, ObsPt) ∈ Properties do
2      ObsVar ← variables of Formula;
3      CF ← compiler fence associated with ObsPt;
4      for OV ∈ ObsVar do
5          ReachDefs ← set of reaching definitions of OV for ObsPt;
6          PriorDefs ← set of definitions of OV preceding any reaching
             definition in ReachDefs in program order;
7          for PD ∈ PriorDefs do
8              insertArtificialDefinition(PD, PD, After);
9          for RD ∈ ReachDefs do
10             insertArtificialDefinition(RD, RD, After);
11         NextDefs ← set of definitions of OV following ObsPt in program
             order;
12         for ND ∈ NextDefs do
13             for O ∈ operands of ND do
14                 insertArtificialDefinition(O, ND, Before);
```

These additional precautions are implemented in Algorithm 2. This time, all definitions of "sibling" variables are considered, through the iteration over all definitions stemming from a given *source variable* in the functional property via renaming

[1] This is related to the program dependence web in classical compiler texts (Aho et al., 2006).

and live range splitting. Notice the "program order" requirement (lines 8 and 14) that prevents reordering of live ranges.

```
1  %k2 = xor %k1, %m
2
3  ...
4  %mpt1 = call get_val1()
5
6  ...
7  %mpt2 = call get_val2()
8
9  %tmp = xor %k2, %mpt2
10
11 // Prop: %tmp == %k2 ^ %mpt2
12
13
14 %k3 = xor %tmp, %m
15 ...
16
17 %mpt3 = 42
18 ...
```

(a) Original

```
1  %k2 = xor %k1, %m
2  %k2' = call artificial_def(%k2)
3  ...
4  %mpt1 = call get_val1()
5  %mpt1' = call artificial_def(%mpt1)
6  ...
7  %mpt2 = call get_val2()
8  %mpt2' = call artificial_def(%mpt2)
9  %tmp = xor %k2, %mpt2'
10 %tmp' = call artificial_def(%tmp)
11 call obs_pt("%tmp' == %k2' ^ %mpt2'")
12 %tmp'' = call artificial_def(%tmp')
13 %m' = call artificial_def(%m)
14 %k3 = xor %tmp'', %m'
15 ...
16 %42 = call artificial_def(42)
17 %mpt3 = %42
18 ...
```

(b) Solution

Listing 4.5: Preserving functional property considering debug information.

Applying the revised algorithm to Listing 4.5a yields the code in Listing 4.5b. Let us illustrate its application for the observed variable `mpt` from the source program. We use debug information to retrieve all definitions of `mpt` in the IR, with `%mpt2` being the reaching definition for the observation point. To prevent any preceding definition (i.e. `%mpt1`) from being moved after `%mpt2`'s definition, Algorithm 2 inserts artificial definition (of `%mpt1'`) right after the definition of `%mpt1` (line 4 from Listing 4.5b). then artificial definition of `%mpt2'` is inserted right after the definition of `%mpt2`. `%mpt2'` also ensures that the reaching definition of the observed variable (`%mpt2`) cannot be optimized out (line 9 from Listing 4.5b). Similarly, artificial definitions are inserted to prevent all succeeding definitions of the observed variable from being moved up before the observation point (lines 12, 13 and 16 from Listing 4.5b): e.g. `%42`'s definition is placed right before `%mpt3`. According to the use-def property, and as the relative order of artificial definition (e.g. `%mpt1'`, `%mpt2'`, `%42`) and the observation point cannot be altered (the I/O side-effecting property of artificial definitions and observation points, cf. Section 4.2), it is guaranteed that the correct value of the observed variable `mpt` will always reach the observation point (via the variable `%mpt2'`). Helper function *insertArtificialDefinition* was described with Algorithm 1 in Section 4.2; it is extended here to operate either on instruction operand or definition of a variable, be it SSA variable or not.

4.3.4 Functional Properties in LLVM

Let us now introduce our modifications to LLVM to support property preservation. Figure 4.1 gives an overview of the augmented framework. All developments took place in the latest, continually updated version of clang and LLVM.

4.3.4.1 Functional Properties in LLVM IR

As presented in Section 3.1.3, the LLVM IR allows metadata to be attached to instructions, to convey additional information to optimizers and code generators. In

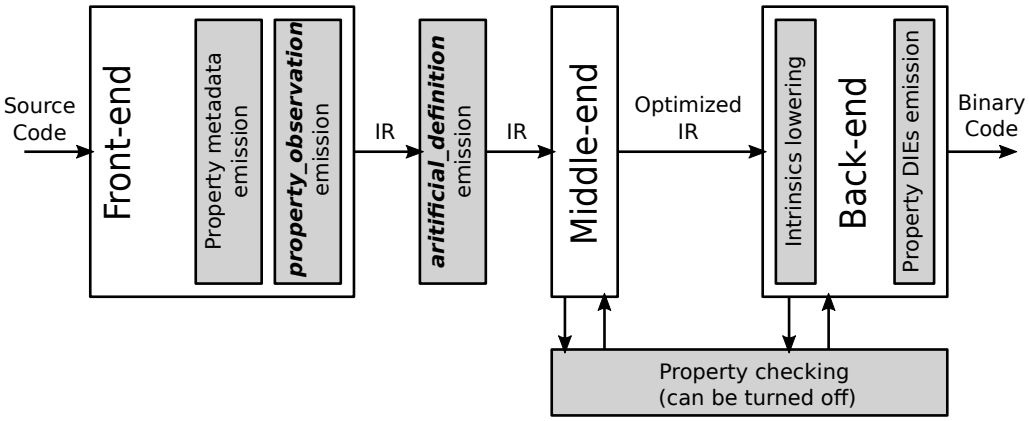

FIGURE 4.1: Overview of the compilation flow extensions. Grey boxes represent new components.

particular, LLVM debug information is implemented as metadata. On the one hand, it is generally maintained and updated by optimization passes. On the other hand, as already mentioned, we want to emit functional properties as DWARF DIEs in the binary. Therefore, it is natural to expand debug information metadata to represent properties in the IR. To this end, we introduce a new type of metadata containing the formula, to which we attach classical debug metadata representing the observed variables and memory locations.

We created an LLVM library that parses our annotation language, which is used by an extension of clang front-end to generate, from the GNU annotation attributes, the appropriate property metadata. More specifically, clang first creates a property metadata containing the property string as well as the information documenting where the property is declared in the source file (i.e. *(file, line, column)* triplet). Next, the property string is parsed to build the list of observed variables and memory locations. This is in turn used to gather debug metadata corresponding to these observed entities, which is finally attached to our property metadata.

We introduced two new intrinsics to the IR: `property_observation` representing an observation point, as well as `artificial_definition` which acts as a live range splitting mechanism constraining optimizations across the observation point.

Observation joint intrinsics: As explained in Section 4.2, `property_observation` embeds a compiler fence to guarantee the correctness and ordering of the values stored in the observed memory locations. In addition, it is declared as having side-effects, so that it cannot be optimized out by optimizations.

Naturally, property metadata will be attached to the corresponding `property_observation`. However, as previously pointed out in Section 3.1.3, a subtle point about metadata is that while optimizations strive to maintain it—debug information in particular, it is always safe to discard it without affecting correctness. Dropping debug information metadata just means that less information is available to the debugger, but it does not invalidate any remaining debug information. When it comes to transmitting functional properties, we need to ensure the metadata presence and its correctness even at the cost of losing some optimizations. We have to make sure that no optimization removes metadata representing functional properties, observed variables or memory locations.

To this end, instead of attaching the property metadata to a given `property_observation`, we make it an *operand* of this I/O side-effecting instruction, which

in turn guarantees that property metadata is also maintained during compilation. `property_observation` intrinsics are inserted to the program by clang, at the same time as the latter creates property metadata from the GNU annotation attributes.

Artificial definition intrinsics: Given these observation points, we implemented Algorithm 2—which creates and inserts `artificial_definitions` to the program—as an IR pass. These intrinsics are also I/O side-effecting, so that their order—relative to each other and relative to the observation point—is preserved by optimizations. Our IR pass is scheduled to run before any optimizations, and thus works on LLVM IR produced by clang, which is not in SSA form yet. This is because the LLVM SSA construction pass (`mem2reg`) is run only after a few other optimizations, at which point the pristine partial state at the observation point defined by a source-level annotation may already be compromised.

4.3.4.2 Functional Properties in LLVM MIR

Later in the LLVM back-end, we modified the "instruction selection" and "MIR formation" phases, such that our intrinsics are lowered into pseudo-instructions with the same semantics. More specifically, property observation pseudo-instructions behave as memory barriers and also have other I/O side-effects, while artificial definition pseudo-instructions are only I/O side-effecting. Unlike other pseudo-instructions, these are not expanded and do not emit any machine instruction, instead they are completely removed during code emission. Moreover, to guarantee the correct functional behavior of the generated program when removing these pseudo-instructions, and to avoid generating additional costly `mov` instructions (which also increases register pressure during "register allocation"), the artificial definition pseudo-instructions use the same source and destination register.

Finally, we modified the "code emission" phase of the compiler back-end so that property DIEs are built from the property metadata, and emitted into the object file's debug section. Moreover, we also modified LLVM's DWARF reader library to support parsing and reading the property DIEs, which will be used by property consumers such as debuggers or binary analysis tools. More precisely, the reader provides an interface to communicate the information about the property to its consumer: it retrieves the machine code address corresponding to the observation point, and also reports the location information of all observed variables and memory locations of the property at this point.

Additionally, we implemented a mechanism to verify the presence and sanity of functional properties throughout the compilation flow. Before performing any optimization, we insert a LLVM pass that registers all properties within the program into the metadata section. Then, after each optimization pass, we insert a verification pass checking the presence of the metadata representing the property, its observed variables and memory locations. A warning informs the programmer if any verification pass fails; she may react by annotating the program differently, or disabling the optimization. This optional mechanism is only used for validation purposes and is disabled in our evaluation process.

On a side note, let us discuss about the interaction of our implementation with *Link-Time Optimization* (LTO) technique, i.e. program optimizations during linking. In a nutshell, the linker—which pulls all object files together and combines them into one program—can see the whole of the program and can therefore do whole-program analysis and optimization. Modern compilers supporting LTO (such as

LLVM and gcc) usually generate special object files which also contain its intermediate representation of the program. Then the linker, extended via a plugin in general, interfaces with the compiler to provide it with information about the files to be linked, which in turn allows for the whole code base to be optimized. For LLVM, all of this process happens on the LLVM IR, where our inserted intrinsics are still available. As a result, our implementation is fully compatible with LTO technique.

4.4 Experimental Validation

We now present the experimental methodology that we used to validate our I/O-barrier-based mechanism and its implementation, followed by functional validation and applications to security properties. We then analyze the impact of preserving source-code protections on the program performance and compilation time.

4.4.1 Methodology

Property preservation is defined as the equality of observation traces (cf. Definition 4.1.6 (Functional property preservation)). Our validation approach is based on comparing, for the same input data, the observation trace produced when executing the binary compiled with our property-preserving compiler against a *reference observation trace*. Both observation traces are obtained using the debugger gdb version 8.3.

To obtain the reference trace, we instrument the original program *without properties*: instead of inserting annotations representing program properties, we introduce C labels to model equivalent observation points, then later set breakpoints at these labels in the debugger. We further mirror all the variables and memory locations that should be observed into specially-named variables and dump their values, at the breakpoints registered above, using the debugger. We compile the instrumented program with optimizations disabled (`-O0`), so that the C labels are preserved, while the mirrored variables are not optimized out. By doing so, we assume the debug information generated by the compiler in the absence of optimization is correct, and that `-O0` preserves the state of the ISO C abstract machine. However, this is not the case in general, as C does not fully specify the ordering of commutative and associative operations, evaluation of function arguments, etc. We mitigate these ambiguities when generating the reference observation trace by linearizing, in the source program, the expressions involved in functional properties to three-address form. Note that instrumentation mirroring is only used for functional validation; it is not activated in any performance or compilation time measurement.

For all other observation traces, we use the modified DWARF reader to retrieve the addresses of the observation points (and set breakpoints at these points in the debugger), as well as the location information (constant value, register number or memory address) of the observed variables and memory locations. The binary is then executed and the values of the observed variables and memory locations at the different breakpoints are retrieved using the debugger.

4.4.2 Functional Validation

Let us now survey the validation of our approach. This consists in first validating the correctness of our mechanism using the methodology described in Section 4.4.1, then leveraging our mechanism to augment the analysis process of binary code.

4.4.2.1 Validating Mechanism Correctness

We first validate our implementation on the test suite of *Framework for Modular Analysis of C programs* (Frama-C) (Cuoq et al., 2012), a reference source code analysis platform for C. Properties are written in *ANSI/ISO C Specification Language* (ACSL) (Baudin et al., 2008) as program annotations. The test suite is designed to validate different Frama-C analyses on a range of small C programs representative of the language semantics. We restrict ourselves to boolean expressions as functional properties, ignoring test cases referring to more advanced ACSL built-in constructs. This results in 30 applicable test cases featuring 558 functional properties. Most of these properties verify the expected values of different variables at a given program point. These test cases, being small programs designed to validate the correctness of Frama-C static analyses, are not meant to be evaluated as performance benchmarks. Therefore, we only use them to validate the correctness of our implementation.

For the correctness validation, we target the x86-64 instruction set and compile each of these test cases at 5 optimization levels `-O1`, `-O2`, `-O3`, `-Os`, `-Oz`. Using the methodology described in Section 4.4.1, we verified that all 558 properties have been correctly propagated to the binaries and produce identical observation traces to the reference one, for all 5 optimization levels considered.

4.4.2.2 Automating Binary Analysis

We also illustrate the applicability of our functional property preservation mechanism in the context of binary analysis. As mentioned in Section 1.1.1, Section 2.1.2 and Section 2.1.3, different analysis tools require additional properties in order to carry out their analyses on executable binaries. For example, robustness analysis framework, such as RobustB (Bréjon et al., 2019), evaluates the robustness of a code region against fault attacks by verifying that a given property, associated to a code region, holds when the latter is submitted to a fault attack.

More specifically, RobustB combines symbolic execution, static analysis and formal verification to find vulnerabilities in all the possible executions of a binary program subject to fault attacks. It models the vulnerabilities detection as formal equivalence-checking formulas that are solved by a SMT solver. The input of the framework consists in a binary code with the target region to be analyzed, together with a property to be verified at the end of the analyzed region. In general, the so-called security property describes the correct behavior of the code region in presence of injected faults, at the end of the region. As a result, it can be expressed using functional property, as shown in the example from Section 1.1.1.

We have combined our compilation framework with RobustB in order to automate the communication of the property required to performed the robustness analysis. We developed a new module to RobustB which basically consists of (1) our modified DWARF reader and (2) our annotation parser library. RobustB uses the former to retrieve the property DIE from the binary debug section, which contains the property string, while the latter is used to parse the string into a list of observed entities. The DWARF reader is then used again to determine the memorizing elements associated to the observed entities at the observation point, reported in the DIEs representing these entities. Using this information, RobustB will be able to build its logic formula that is further verified by a SMT solver. The verification is either *satisfiable*—which means that a vulnerability invalidating the considered property is detected—or *unsatisfiable*—implying that the fault has no effect on the program,

or it is caught by a fault detection mechanism. We target the ARMv7-M/Thumb-2 instruction set, which is representative of deeply embedded devices.

We validated our automated robustness analysis framework on different implementations of the PIN authentication from the benchmark suite dedicated to fault injection analysis (Dureuil et al., 2016), compiled at different optimization levels. The target code region is a function implementing an authentication service through the comparison of an user-provided PIN code against the PIN code of the card. It returns a value indicating whether the access is granted for a given user-provided PIN. To carry out the analysis, we considered the vulnerability expressing the invalidation of the property described in Section 1.1.1: it describes the scenario where the function returns the value granting the access while the entered PIN code is incorrect. This experiment (1) shows that our mechanism allows to maintain and propagate functional properties throughout aggressively-optimizing compilation flow, and (2) highlights the relevance of our property-preserving compilation framework to improve the analysis flow performed on executable binaries.

4.4.3 Preserving Security Protections

Let us now consider the security use-cases described in Section 3.3 and illustrate how we can leverage our property-preserving mechanism to preserve security protections.

Recalling that applications are commonly secured by inserting protections at the source code level, however, compilers may not understand the programmers' intentions, missing the implicit link between secure code protections and the control flow or machine state assumptions underlying its function, optimizing the protection away as a result. As previously stated, programmers devise complex coding tricks to obscure their intentions, but compiler engineers find smarter ways to optimize code. As a result, compilers can optimize away the security protections that they consider as doing no "useful" work.

Instead, we argue that programmers should be able to instruct compilers to preserve the protections and enforce the associated security properties by encoding the protection-derived property as functional property, which will then be propagated and preserved throughout the compilation flow, even in the presence of optimizations. More specifically, we will illustrate our idea by considering 4 use-cases, presented in Section 3.3, covering the following security protection-derived properties, necessary to the effectiveness of the associated countermeasures:

- secret erasure property (i.e. proper erasure of sensitive data in memory, cf. Section 3.3.1);
- instruction ordering property (i.e. proper instruction ordering in masked secret key operations, cf. Section 3.3.2);
- code interleaving property (i.e. proper fine-grained interleaving of functional and protection code, cf. Section 3.3.3);
- redundancy preservation property (i.e. presence of redundant data and code to detect fault injections, cf. Section 3.3.4).

To make our case, *we thus need to encode the implicit assumptions underlying code hardening techniques using functional properties of the source program*. The key observation that hints at such smart functional encoding is that, for the considered security protections, which all introduce new variables to the program, preserving

these variables at some specific program point and ensuring their uses in the code subsequent to that point suffice to ensure the preservation of the countermeasure schemes. The only exception is SCI protection, which will be later explained in Section 4.4.3.3. Our I/O-barrier-based mechanism allows preserving variables that are part of partial states defined by functional properties. This means that it is possible to preserve security protections by first encoding the protection-derived properties as functional properties that refer to key variables of these protections, then leverage our I/O-barrier-based mechanism to preserve these variables, thus preserving the associated protections. As pointed out in Section 4.1, these properties do not necessarily encode any logical formula, and could be naturally expressed as the special type of functional property, namely observational properties (c.f. Definition 4.1.4 (Observational property)).

In order to express observational properties, we extend the annotation language with a minimalistic predicate called `observe()`. This predicate does not encode any logical formula; it takes the variable to be observed as argument and simply includes it into the partial state defined by the property and thus into the program observation trace. In other words, `observe()` provides a convenient interface to explicitly specify the observed variables that need to be preserved by the compiler at the observation point.

In the following, for each security use-case, we detail how to encode the security protection-derived property associated to the source-level protection scheme as an observational property using the predicate `observe()`. We verify the property preservation with our approach by comparing traces for all optimization levels `-O1`, `-O2`, `-O3`, `-Os`, `-Oz`, following the methodology previously described in Section 4.4.1. Again, similarly to Section 4.4.2.2, we target the ARMv7-M/Thumb-2 instruction set, which is representative of deeply embedded devices. For the trace generation, we emulate the execution of the applications with the QEMU emulator version 3.0.1, monitored by the cross-compiled debugger gdb version 8.3.

4.4.3.1 Sensitive Memory Data Erasure

Let us now consider the scenario, described in Section 3.3.1, where a sensitive buffer on the stack must be zeroed with a call to `memset()` to avoid leaking confidential information in a cryptographic application. The code snippet shown in Listing 4.6a illustrates the scenario.

```
1
2
3  ...
4  void process_sensitive(void) {
5    uint8_t secret[32];
6    ...
7    memset(secret, 0, 32);
8
9  }
```

(a) Original attempt

```
1  #define ANNOT(s)
2    __attribute__((annotate(s)))
3  ...
4  void process_sensitive(void) {
5    uint8_t secret[32];
6    ...
7    memset(secret, 0, 32);
8    obs_pt: ANNOT("observe(secret)");
9  }
```

(b) Using annotation

Listing 4.6: Example of sensitive memory data erasure.

In order to guarantee the secret erasure property of `secret`, we insert an observational property right after zeroing, before the function returns, as illustrated in Listing 4.6b. The `observe` property instructs the compiler to insert an observation point acting as a memory barrier (cf. Section 4.2), which in turn forces the effective zeroing of the whole secret buffer even with optimizations enabled. Note that

not only the compiler inserts a memory barrier to enforce the presence of `memset()` in the program, but the address of the `secret` buffer will also be attached into the property metadata, which is later emitted into the property DIE. The latter provides an easy mean to retrieve the information, needed for generating observation traces during trace comparison. This is a simple example of combining hardening code—by inserting a security protection (erasure of the buffer `secret`)—and an observational property—encoding the protection-derived property which allows for preserving the protection (observing the content of `secret`)—to enforce a security property (leakage prevention).

By comparing the reference trace and observation traces obtained by executing the binary compiled at all optimization levels (cf. Section 4.4.1), we verified that the whole buffer `secret` was correctly zeroed at the observation point, in all observation traces.

4.4.3.2 Masking Computation Order

Let us now consider the scenario of mask swapping. Masking of secret value seeks to ensure that all computations involving this value are statistically independent of it, thus avoiding leaking confidential information in a cryptographic application. To further maintain this statistical independence from the secret value, notably when only a limited set of masks is available, mask swapping is introduced to the program (cf. Section 3.3.2). The code snippet shown in Listing **??** illustrates the scenario.

The effectiveness of this mask swapping scheme requires that the re-masking operation has to take place before the removal of the previous mask. Let us now illustrate how our mechanism can be used to force the compiler to respect this specific computation order.

```
1
2
3 ...
4
5
6 round_key[i] = (k[i] ^ mpt[i]) ^ m;
```

(a) Original attempt

```
1 #define ANNOT(s)
2   __attribute__((annotate(s)))
3 ...
4 uint8_t tmp = k[i] ^ mpt[i];
5 obs_pt: ANNOT("observe(tmp)");
6 round_key[i] = tmp ^ m;
```

(b) Using annotation

Listing 4.7: Respecting computation order in mask swapping operation.

Listing 4.7b shows our proposed solution to guarantee the instruction ordering property. We first linearize the original expression to three-address form by defining a temporary variable `tmp` to hold the result of re-masking (line 4), then later use it for unmasking (line 6). We also insert the observational property `observe(tmp)` between re-masking and unmasking operations (line 5), which forces the compiler to preserve the value of `tmp` at the observation point, and use this exact value in the subsequent unmasking operation.

For this use-case, the reference observation trace is generated by compiling, at `-O0`, the version shown in Listing 3.5, which leverages coding trick involving volatile-qualified temporary variable together with memory barrier implemented using inline assembly in order to guarantee the proper instruction ordering of re-masking and unmasking operations. By comparing the reference trace and observation traces obtained by executing the binary compiled at all optimization levels (cf. Section 4.4.1), we verified that the temporary variable at the observation point did hold the expected result of the re-masking operation, in all observation traces.

4.4.3.3 Step Counter Incrementation

We now consider the SCI countermeasure scheme that aims at detecting fault attacks that induce unexpected jumps to any location in the program (i.e. not executing at least two adjacent statements in of C source programs). The protection defines a step counter at each control construct, and steps the counter of the immediately enclosing control construct after every C statement of the original source. Counters are regularly checked against their expected values, calling a fault handler when it fails (cf. Section 3.3.3). An example of the SCI technique is illustrated in Listing 4.8a.

As a reminder, in order for the SCI countermeasure scheme to be effective, one needs to ensure the proper interleaving of original statements and counter incrementations and checks—i.e. ensure the code interleaving property—which, unfortunately, will be altered by compiler optimizations. In fact, as explained in Section 3.3.3, the compiler will be able to deduce the constant values of the counters `c_then` and `c_else` at every check, thus optimize away these checks as it has no notion of fault injection. As a consequence, the results of counter incrementations are not needed in the program anymore, making incrementations ideal candidates for being removed next.

```
1
2
3  ...
4  unsigned short c_then = 0;
5  unsigned short c_else = 0;
6  if (cond) {
7
8
9
10   if (c_then != 0) killcard();
11   c_then++;
12   a = b + c;
13
14
15
16   if (c_then != 1) killcard();
17   c_then++;
18 }
19
20
21
22 if (!((c_then == 2 &&
23         c_else == 0 && cond) ||
24        (c_then == 0 &&
25         c_else == 0 && !cond)))
26   killcard();
```

(a) Original countermeasure scheme

```
1  #define ANNOT(s)
2    __attribute__((annotate(s)))
3  ...
4  unsigned short c_then = 0;
5  unsigned short c_else = 0;
6  if (cond) {
7  obs_pt1: ANNOT("observe(
8      c_then,
9      c_else, cond, a, b, c)");
10   if (c_then != 0) killcard();
11   c_then++;
12   a = b + c;
13 obs_pt2: ANNOT("observe(
14     c_then,
15     c_else, cond, a, b, c)");
16   if (c_then != 1) killcard();
17   c_then++;
18 }
19 obs_pt3: ANNOT("observe(
20     c_then, c_else, cond,
21     a, b, c)");
22 if (!((c_then == 2 &&
23         c_else == 0 && cond) ||
24        (c_then == 0 &&
25         c_else == 0 && !cond)))
26   killcard();
```

(b) Using annotation

Listing 4.8: Preserving code interleaving in SCI.

Consider Listing 4.8b, which shows our proposed solution to preserve the protection scheme. First, we protect counter incrementations and checks by inserting observational properties before every check (lines 7, 13 and 19). Each property observes the variables involved in the evaluation of the associated check: `c_then` (lines 8, 14 and 20), `c_else` and `cond` (line 20). This implies the automatic insertion of artificial definitions taking as parameter the original values of these variables and producing the opaque values that replace the original ones in the subsequent code, i.e. the associated check (cf. Section 4.2). As a result, the values of these observed variables are

hidden from the compiler when evaluating these checks. Moreover, in order to guarantee that the original and countermeasure instructions are correctly interlaced, we leverage the barrier effect at the observation point of functional and observational property with regards to accesses to observed memory locations and definitions of observed variables (cf. discussion on Definition 4.1.3 (Functional property)). To this end, the inserted properties also observe all other program variables and memory locations containing valid values (lines 9, 15 and 21). In other words, we have to observe the *complete* program state instead of a partial one, in order to ensure that no other modifications to the complete program state have been introduced between two given consecutive observational properties. We directly modified the macros implementing the countermeasure schemes from the original work (cf. Section 3.3.3) to insert our annotations before each check.

We verified at all optimization levels that all values are correct with respect to their counterparts from the reference observation trace. However, for the `sci-aes` benchmark, the debugger was unable to retrieve some values (all other values were correct), always less than 0.2% (e.g. 9907 out of 8353591 at `-O2` and `-O3`). The unavailable values are due to various bugs in LLVM's back-end, which generate, incorrect location information in the debug information describing the observed variables. For example, the instruction scheduling pass may misplace or worse, remove the debug pseudo-instructions which provide information when a source variable is set to a new value. We have tried to fix some of these bugs, and reported some others[2,3]. Since it is not feasible to find and fix all the bugs affecting the debug information correctness in LLVM, we did manually verify the presence of the observed variables for all these unavailable values: indeed, while they do have their expected values stored in a register or in memory, gdb does not know about it and reports the variables as optimized out. This confirms that our mechanism does actually maintain and propagate properties annotated in the source program, down to machine code.

4.4.3.4 Control and Data Flow Redundancy

Finally, we illustrate the preservation of the loop hardening scheme by encoding the protection-derived property as observational one. The loop protection consists in duplicating termination conditions and the computations involved in the evaluation of such conditions (cf. Section 3.3.4), in order to ensure that the hardened loop always performs the expected number of iterations and takes the right exit, even in the presence of fault attacks. We consider an implementation of the source-level loop protection on `memcmp()` function, shown in Listing 4.9a.

Listing 4.9b illustrates our proposed solution to preserve the redundancy preservation property during optimizing compilation. To prevent optimizations from removing the checks involving the duplicated loop counter `i_dup` (line 11 and 22) and the duplicated loop-independent variable `n_dup` (lines 17 and 25), we observe the duplicated variable involved in the check, right before the latter (lines 10, 16 and 24). Similar to the previous example, the values of the duplicated variables are hidden from the compiler when evaluating these checks (thanks to the opaque values produced by artificial definitions, which are guaranteed to replace the original values in these checks), thus preserving the latter.

We verified at all optimization levels that the duplicated variables at the observation points did hold the expected value, as found in the reference observation trace. This confirms the preservation of the duplicated variables down to machine code.

[2]https://bugs.llvm.org/show_bug.cgi?id=37391
[3]https://bugs.llvm.org/show_bug.cgi?id=37149

```
1
2
3  ...
4  int memcmp(char *a1, char *a2,
5             unsigned n) {
6    unsigned i, i_dup, n_dup = n;
7    for (i = 0, i_dup = 0;
8         i < n;
9         ++i, ++i_dup) {
10
11     if (i_dup >= n)
12       fault_handler();
13     if (a1[i] != a2[i]) {
14       if (a1[i_dup] == a2[i_dup])
15         fault_handler();
16
17       if (n_dup != n)
18         fault_handler();
19       return -1;
20     }
21   }
22   if (i_dup < n)
23     fault_handler();
24
25   if (n_dup != n)
26     fault_handler();
27   return 0;
28 }
```

(a) Original attempt

```
1  #define ANNOT(s)
2    __attribute__((annotate(s)))
3  ...
4  int memcmp(char *a1, char *a2,
5             unsigned n) {
6    unsigned i, i_dup, n_dup = n;
7    for (i = 0, i_dup = 0;
8         i < n;
9         ++i, ++i_dup) {
10 obs_pt1: ANNOT("observe(i_dup)");
11     if (i_dup >= n)
12       fault_handler();
13     if (a1[i] != a2[i]) {
14       if (a1[i_dup] == a2[i_dup])
15         fault_handler();
16 obs_pt2: ANNOT("observe(n_dup)");
17       if (n_dup != n)
18         fault_handler();
19       return -1;
20     }
21   }
22   if (i_dup < n)
23     fault_handler();
24 obs_pt3: ANNOT("observe(n_dup)");
25   if (n_dup != n)
26     fault_handler();
27   return 0;
28 }
```

(b) Using annotation

Listing 4.9: Preserving redundancy in loop hardening.

4.4.4 Performance and Compilation Overhead Evaluation

Let us now analyze the run-time performance and compilation overhead for all considered security applications. We compare our versions with binaries generated with (1) no optimizations at all—which is also a solution to preserving security protections, (2) with the common-practice (unreliable) programming tricks to prevent the compiler from removing source-code protections (when available) and (3) to set an upper bound on the achievable performance, with the insecure binaries compiled without any property preservation mechanism. We target the ARMv7-M/Thumb-2 instruction set (Cortex-M3 embedded processor). The performance results are obtained using ARM Fast Models (ARM, 2019).

4.4.4.1 Performance

We measure the program performance in terms of number of instructions executed in the main program, using ARM Fast Models (ARM, 2019). In fact, this is a relevant choice, given its consistency on the Cortex-M3's very simple pipeline. Indeed, as will be shown in Section 5.6.1, when comparing program performance for Cortex-M3 processor, measurements of execution time (in number of clock cycles) give the same results as measurements based on number of instructions executed. For each application, Figure 4.2 presents the performance ratio of different versions compiled at different optimization levels with respect to the original program compiled at `-O0`, which serves as a baseline. The first version corresponds to the original code without any modification, the second includes programming tricks described per security use-case in Section 3.3, and the last one has source-code annotations inserted as described in Section 4.4.3. These three versions are referred as `Insecure`,

`Tricks` and `Annotations` respectively in Figure 4.2. It is worth noting that there are no programming tricks to reliably preserve SCI and source-level loop hardening.

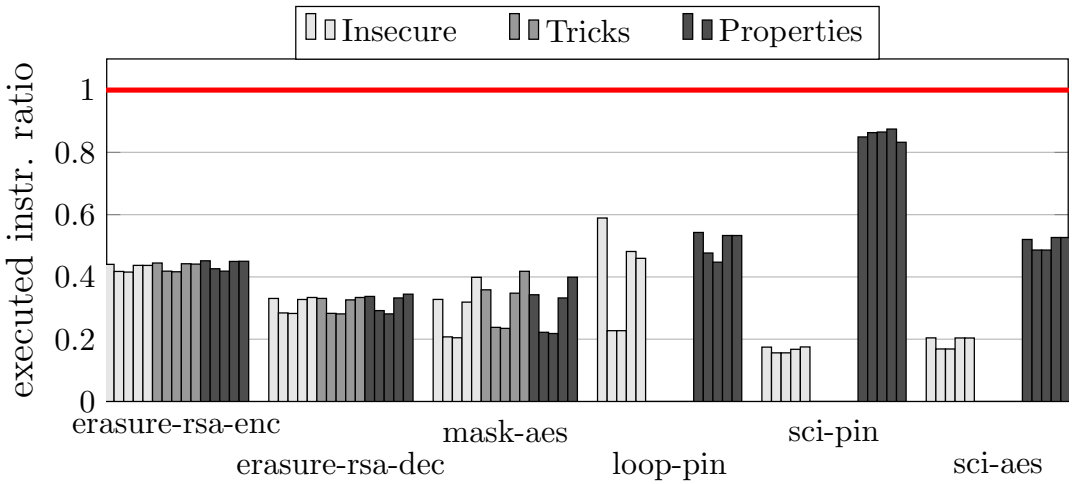

FIGURE 4.2: Executed instructions w.r.t. `-O0` baseline (horizontal red line), ordered by optimization level `-O1`, `-O2`, `-O3`, `-Os`, `-Oz`

Results show that (1) the insecure versions yield to the fastest (but non-secure) executables, since protections are modified or removed with optimizations enabled; (2) compared to existing programming tricks, our compiler preserves the source-level protections with similar, if not better performance (furthermore, programming tricks are not a viable solution, as they are neither portable, nor future-proof); (3) when no trick exists (`loop-pin`, `sci-pin` and `sci-aes`), our compiler provides consistent performance improvement over programs compiled at `-O0` while preserving the source-level protection. The higher cost of preserving the protection of `sci-pin` compared to the one of `sci-aes` can be traced back to the functional/protection code ratio for these 2 programs. Indeed, `sci-pin` features about 4 times more protection code than functional code, with the functional code being too trivial to be optimized within the boundaries of the protection statements; while `sci-aes` contains only twice more protection code than functional code. Moreover, the functional code of `sci-aes` is more complex, leaving potential for classical optimizations. Finally, note that, as security engineers are in search of a mechanism to reliably preserve source-level security protections throughout the compilation, performance is not the primary motivation in these examples: anything better than `-O0` is beneficial to security engineers.

4.4.4.2 Compilation Time

Figure 4.3 shows the compilation-time overhead for all applications, compared to the original program (without annotations), compiled with the same optimization flag. The validation platform is a quad-core 2.5 GHz Intel Core i5-7200U CPU with 16 GB of RAM. In general, the compilation overhead is under 10%, though it can sometimes be really important, when complete program state is constantly observed, as in the case of the `sci-pin` and `sci-aes` benchmarks. Furthermore, the SCI protection scheme introduces a large number of observation points (at least one annotation for every functional C statement). As a result, this is really a worst case scenario. In other words, the compilation-time overhead depends on the intended protections. As noted earlier, security community is searching for the reliable preservation of source-level security protections down to executable binary; the community thus is,

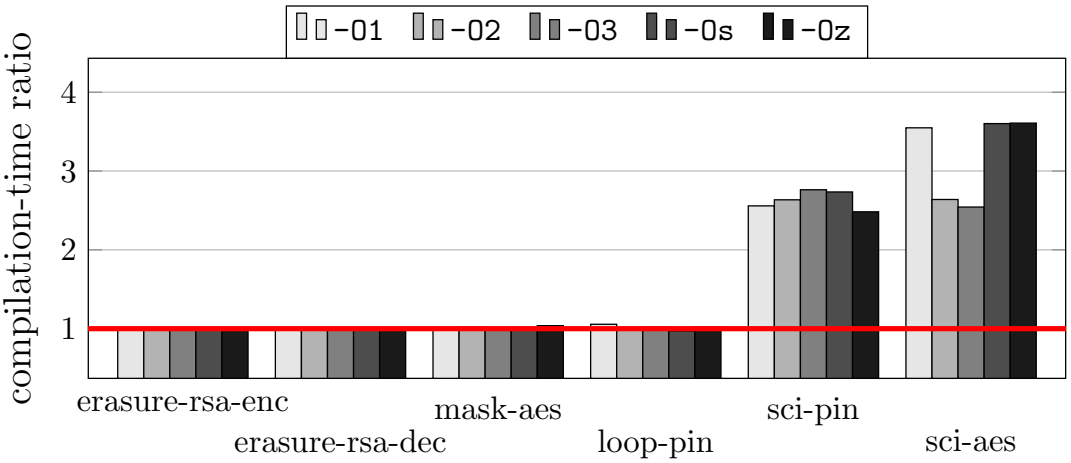

FIGURE 4.3: Compilation-time w.r.t. original program without functional property annotations (horizontal red line), ordered by optimization level `-O1`, `-O2`, `-O3`, `-Os`, `-Oz`.

in general, willing to pay the price of compilation time in exchange. Moreover, our prototype is not yet tuned and optimized and we believe that compilation-time can be reduced with additional algorithmic and engineering effort, which will be left for future work if necessary.

4.5 Discussion

In this chapter, we defined the notion of functional property preservation and proposed an I/O-barrier-based approach to translate and preserve functional properties across all program representations through the optimizing compilation of C code to machine code, such that optimizations will preserve the consistency of these properties. We implemented and validated our approach in the LLVM framework, with no changes to existing optimization passes beyond bug fixes related to the propagation of debug information. While the problem we consider may have general applications in software engineering (as described in Chapter 1 and Chapter 2), our proposal specifically addresses the fundamental open issue in security engineering of preserving security countermeasures introduced at source program down to machine code. To this end, we extend the notion of property preservation to include observational properties, which can be seen as a special case of functional properties.

Our I/O-barrier-based approach relies on the insertion of opaque and I/O side-effecting instructions that the compiler cannot analyze, so that they cannot be removed by optimizations. This also implies the preservation of the order of these instructions, relative to each other and relative to other I/O side-effecting instructions of the original program. However, this ordering preservation is not required by the preservation itself, and not even always desirable, as it introduces additional constraints to compiler optimizations, which might hamper the generated code's performance. Furthermore, since our implementation takes full advantage of the existing debug information collected and generated by LLVM, which does not currently handle the scenario where multiple live ranges corresponding to the same source variable overlap, we have to insert additional I/O side-effecting instructions to avoid such overlap. This inevitably inhibits more compiler optimizations, thus making the generated code non-optimal.

In the next chapter, we argue that such restrictions can be lifted with a more fine-grained approach, solely leveraging the most essential information modeled by almost every compiler transformation, namely data dependence. The programmer will be granted a means to explicitly specify the wanted observational property in the form of such a dependence, that is guaranteed to be preserved by compiler transformations.

Chapter 5

Source-Level Directives for Preserving Property

Chapter 4 provides two important observations: (1) for compilers, preserving source-level functional properties—which, by definition, takes the form of assertion-like boolean expressions that refer to some entities (variables or memory locations) from the source program—can be reduced to preserving the expected values of these entities at program points equivalent to where the properties occur in the source program; and (2) the problem of security countermeasures preservation can be practically addressed by first devising smart encoding of the protection-derived properties associated with the countermeasures, which turns these properties into a special type of functional properties—i.e. observational properties—involving important variables from the countermeasures, and then preserving these observational properties. As a result, the real problem we actually try to address is how to preserve observational properties, or more precisely, how to preserve specific values, that we refer to as observed values, at specific points of the program execution, i.e. the so-called observation point.

To this end, we proposed an I/O-barrier-based solution in Chapter 4. For a given observational property, we first represent the associated observation point with an I/O side-effecting memory barrier—the compiler sees a memory-reading instruction—in order to preserve the observed values of memory locations referred in the property, and that the observation point cannot be removed by compiler optimizations. However, this introduces additional constraints for the compiler when transforming the program. On the one hand, memory accesses cannot be moved across the observation points. On the other hand, embedding I/O side-effects into the observation point unintentionally establishes a total order of all observational properties, including unrelated ones, i.e. ones observing different values, whether at the same program point or not. Enforcing this total order is unnecessary in general, while it may prevent further optimizations, notably during instruction scheduling.

As for preserving observed values of variables from the property, the key idea behind our proposed solution from Chapter 4 lies in what we call *opacification* technique: given an observed value v, we hide it from compiler optimizations using a special identity function—that we call artificial definition—taking v as its argument. This identity function is *opaque*, which means that the compiler sees it as a function computing a completely unknown value v' from v and assumes no correlation between these two values. As a result, the compiler will consider that all different artificial definitions of the same observed value are distinct. Nevertheless, this is not always desirable: for instance, combining these artificial definitions whenever possible would result in more efficient code and become crucial for performance-critical code. Furthermore, to guarantee that optimizations cannot eliminate the introduced

artificial definitions, we also define these as I/O side-effecting. This constraints further instruction scheduling unnecessarily.

In short, embedding volatile side-effects into the program during compilation is rather constraining and too expensive. However, the I/O side-effects are needed to ensure that observational properties and their observed values will not be eliminated by compiler optimizations. Therefore, the natural question we seek to answer in this chapter is whether it is possible to preserve observational properties solely using the opacification technique and if not, what should be the bare minimum of I/O side-effecting instructions required for effective preservation of observational properties.

To this end, this chapter first defines the notion of observation and its preservation through program transformations and illustrates these on a simplified intermediate program representation. We then describe an *I/O-barrier-free* mechanism to preserve observations down to machine code and formally prove its correctness on our simple IR. Next, we detail our LLVM-based implementation with insignificant modification to individual optimization passes, and validate it on the security use-cases presented in Section 3.3. Finally, we evaluate the performance and compilation time impact of our approach, and further compare with our first solution, as well as with an alternative implementation of our mechanism.

5.1 Problem Definition

Recalling that both functional and observational properties that we seek to preserve throughout the compilation process are external to the program code and thus are primarily not known to the compiler. As a consequence, we need to define the semantics of these properties—which can be reduced to the notion of *observation*—in the compiler IR. To this end, we first introduce the syntax (cf. Section 5.1.1) and operational semantics (cf. Section 5.1.2) of a simple language, called Mini IR, representative of the levels of IR typical of the compilation of imperative languages. Next, we extend its semantics with the notion of observations (cf. Section 5.1.3), then define the notion of program transformation (cf. Section 5.1.4), and notably the concept of *observation-preserving transformation*, which specifies the constraints required for preserving observations (cf. Section 5.1.5).

Our Mini IR models control flow at a relatively low level (linearized three-address code) while supporting the usual memory abstractions, SSA values, intra- and inter-procedural constructs. For the sake of simplicity, we will use it to model not only optimizing compiler IR but also source code and assembly code.

5.1.1 Mini IR Syntax

Figure 5.1 presents the grammar of Mini IR, which will be extended to explain our proposed solution described in Section 5.2.

In Mini IR, control flow is implemented as flat CFG of blocks and branches. Unlike traditional CFG- and SSA-based compilers, such as gcc and LLVM, we use branch and block arguments following continuation-passing style (Appel, 1998). Like in *Multi-Level Intermediate Representation* (MLIR) (Lattner et al., 2020), single-assignment variables are declared and scoped in a region (introduced by a function or macro) and captured in dominated blocks; as a consequence, branch arguments only need to carry SSA values, as opposed to explicitly carrying all live variables. This choice makes use-def chains more uniform across an entire function without

var	::=	*ident*	identifier for an SSA value
expr	::=	*literal*	literal constants
	|	*un-op var*	unary operator
	|	*var bin-op var*	binary operator
	|	*ident* `(` *var** `)`	function application or macro expansion
	|	`io(` *var*, *var** `)`	I/O effect with ordering descriptor
	|	`snapshot(` *var*$^+$ `)`	identity function observing its arguments into a partial state
instr	::=	*expr*	expression with no associated definition
	|	`[` *var*$^+$ `=]` *expr*	define a value from an expression
	|	*ref* `<-` *var*	store a value to a reference
	|	*var* `=` *ref*	load a value from a reference
	|	`mem[` *var* `]` `<-` *var*	store a value to a memory address
	|	*var* `=` `mem[` *var* `]`	load a value from a memory address
	|	`br` *var* `,` *ident* `(` *var** `)`	branch with condition, target block identifier and SSA value arguments
	|	`return(` *var** `)`	return from function or macro
block	::=	*ident* `[` `(` *var** `)]` `:` `[` *instr* `;]`*	block labeled by a unique identifier, composed of arguments and an instruction sequence, branch- or return-terminated
region	::=	`{` *block*$^+$ `}`	return-terminated region with one or more blocks
func-decl	::=	`function` *ident* `(` *var** `)` *region*	function definition
macro-decl	::=	`macro` *ident* `(` *var** `)` *region*	macro definition

FIGURE 5.1: Grammar of our Mini IR. The terminals *literal*, *ident*, *un-op*, *bin-op* are the same as the corresponding C lexical tokens.

implicitly referring to control flow edges, which in turn simplifies our formalization of a happens-before relation later in this subsection.

Note that we did not include indirect branches and calls in the syntax. Supporting these would include making identifiers first class and holding them as additional SSA arguments of branch and call instructions. This however does not impact the following formalization.

When clear from the context, we will write "instruction *expr*" when referring to a instruction defining, assigning or returning a value from an expression *expr*.

5.1.2 Mini IR Operational Semantics

Let us now present the operational semantics of our Mini IR. All expressions and instructions have fairly standard semantics, except for `snapshot`, which is in fact the extension to the operational semantics of our language and will be presented in Section 5.1.3.

As a simplifying assumption, we only consider *sequential*, *deterministic* programs with *well defined behavior*. In particular, similarly to Chapter 4, we avoid cases where the compiler may take advantage of undefined behavior to trigger optimizations. This assumption is consistent with widespread coding standards for secure code. Our formalization also assumes *no exceptions* at the source language level, but precise machine exceptions at the instruction level are supported.

Definition 5.1.1 (Name-Value Domains)**.** Every value manipulated during the execution of a program belongs to one of these four *Name-Value domains*:

- $\mathcal{V}$ is a set of (Var, Val) pairs where *Var* is an SSA variable name (e.g. an SSA value in LLVM IR or a variable in a functional language) and *Val* is the value of *Var*; all uses of *Var* are dominated by an unique definition associating *Var* with its value *Val*;
- $\mathcal{C}$ is a set of (Val, Val) pairs where *Val* is a constant value also standing as the name of the constant;
- $\mathcal{R}$ is a set of (Ref, Val) pairs where *Ref* is a reference name (e.g. a C variable, a reference in a functional language, or a register in a low-level representation) and *Val* is the value referenced by *Ref*;
- $\mathcal{M}$ is a set of (Mem, Val) pairs where *Mem* is a memory address and *Val* is the value stored at *Mem*.

We define an operational semantics for our Mini IR in terms of a state machine, where every IR instruction defines a transition referred to as an *event*.

Definition 5.1.2 (Program State)**.** A *program state* is defined by a tuple $(Vals, \pi)$ with $Vals = V \cup C \cup R \cup M$, where $V \subseteq \mathcal{V}$, $C \subseteq \mathcal{C}$, $R \subseteq \mathcal{R}$, $M \subseteq \mathcal{M}$, and the *program point* π holds the value of the program counter pointing to the next instruction to be executed.

Note that, in contrast to Definition 4.1.1 (Program state) defining a program state with a generic notion of program variable (which may refer to either a source-level denotation, an SSA value, or a register in a low-level representation), we deliberately want to distinguish values from different Name-Value domains in order to provide rigorous definitions of different relations between program events (cf. Definition 5.1.8 (Dependence Relation) for example).

Definition 5.1.3 (Event)**.** An *event* e is a state machine transition, associated with the execution of an instruction i, from a state σ into a state σ'. It is denoted by $e = \sigma \overset{i}{\rightsquigarrow} \sigma'$.

For any given event e, let $Inst(e)$ denote the instruction executed by event e.

Definition 5.1.4 (Program Execution)**.** A program execution E is a (potentially infinite) ordered sequence of program states and events:

$$E = \sigma_0 e_0 \sigma_1 e_1 \sigma_2 \ldots \text{ with } e_0 \text{ a special event defining all constant pairs } (c, c) \in \mathcal{C},$$
$$\sigma_0 \text{ the initial state, and}$$
$$\sigma_k \overset{i_k}{\rightsquigarrow} \sigma_{k+1}, \text{ such that } \forall k > 0,\ i_k = Inst(e_k)$$

When executing a non-branch, non-`return` instruction in a basic block, the next state σ_{k+1} points to the next instruction in the block. Executing a branch instruction makes the next state point to the first instruction of the target block. Executing a `return` instruction makes the next state point to the next instruction following the function call instruction that led to the currently executing function. When executing a function call instruction, the next state σ_{k+1} points to the first instruction of the function's enclosed region.

Unlike a function call, macro expansion takes place in an earlier, offline stage, prior to program execution. The macro's region is expanded in place, with effective arguments substituted in place of the formal ones, renaming the region's local variables and references to avoid conflicts with variables and references of the parent

region, and implementing `return` as copying some of the macro's variables into variables defined in the parent. As a result, macro expansion never occurs on a program execution.

In the following, *Inputs* (resp. *Outputs*) represent the sets of all possible inputs (resp. outputs) of P, and *Executions* is the set of executions produced by P. Given an input $I \in \mathit{Inputs}$, a program P produces an unique execution denoted by $\mathcal{E}[\![P]\!](I)$.

Starting from an initial state σ_0, the execution proceeds with calling the special `main` function, taking no argument and returning no value. Instead the program conducts input and output operations through I/O instructions involving the `io` expression. The input and output of a given program is modeled as a (potentially infinite) list of independent (potentially infinite) sets of values, each set identified with an unique descriptor, the first argument of the `io` expression. Every value in an I/O set is uniquely tagged to distinguish it from any other I/O value from the same set.

The execution of an I/O instruction instantiates an I/O event. Every I/O event reads or writes one or more values. For an I/O event e we note $IO(e)$ its input or output values.

The semantics of P is denoted by $\mathcal{C}[\![P]\!]()$, which is a function computing the outputs sets from input sets. Furthermore, the semantics of P also defines the total ordering of I/O events, as defined below.

Definition 5.1.5 (I/O Relation)**.** Any pair of distinct events associated with the execution of `io` instructions *with the same descriptor* are ordered by a so-called *I/O ordering relation*, denoted by $\xrightarrow{\text{io}}$. Formally, given an execution $E = \mathcal{E}[\![P]\!](I)$ of P on some input $I \in \mathit{Inputs}$, $\xrightarrow{\text{io}}$ is the reflexive and transitive closure of the following relation:

$$\forall \ldots e_1 \ldots e_2 \ldots \in E,\ \Big(\mathit{Inst}(e_1) = \texttt{io}(\mathit{desc}, IO(e_1)) \ \wedge\ \mathit{Inst}(e_2) = \texttt{io}(\mathit{desc}, IO(e_2))\Big) \implies e_1 \xrightarrow{\text{io}} e_2$$

This relation on events induces a relation on values in input and output sets, also denoted by $\xrightarrow{\text{io}}$:

$$\forall \ldots e_1 \ldots e_2 \ldots \in E,\ \Big(\mathit{Inst}(e_1) = \texttt{io}(\mathit{desc}, IO(e_1)) \wedge \mathit{Inst}(e_2) = \texttt{io}(\mathit{desc}, IO(e_2)) \ \wedge\ e_1 \xrightarrow{\text{io}} e_2\Big) \implies IO(e_1) \xrightarrow{\text{io}} IO(e_2)$$

In addition, when a single I/O event reads or writes multiple values, they are ordered from left to right in a given `io` instruction and sequentially over successive `io` expressions associated with the same event.

The $\xrightarrow{\text{io}}$ relation on input and output data models streaming I/O as well as unordered persistent storage in computing systems, and any middle-ground situations such as locally unordered streaming I/O and locally ordered storage operations.

5.1.3 Mini IR Observation Semantics

Let us now consider the last expression in our Mini IR syntax. `snapshot` expressions introduce a specific mechanism to observe values along the execution of the program. In Section 4.1, we defined an *observation trace* as a sequence of program

partial states—sets of (variable, value) and (address, value) pairs. To increase the reach of compiler optimizations while preserving the user's ability to attach logical properties to specific values and instructions, we extend the observation semantics to *partially ordered observation states* defined by the execution of instructions involving `snapshot` expressions.

Definition 5.1.6 (Observation State)**.** Any event involving a `snapshot` expression define an *observation state* of the operational semantics. These observation states are modeled by the following *observation function*:

$$Obs : Events \rightarrow States$$

extracting from an event e an *observation state* $(ObsV, ObsC, ObsR, ObsM, \pi)$ such that π is the program point of the instruction associated with e and $ObsV \subseteq V$, $ObsC \subseteq C$, $ObsR \subseteq R$, $ObsM \subseteq M$ are the $(name, value)$ pairs observed by all arguments of `snapshot` expressions involved in event e.

In addition, an individual instruction involving a `snapshot` expression returns all its arguments in addition to capturing these arguments' $(name, value)$ pairs into an observation state.

Let us now define *observation events* associated with the execution of instructions involving `snapshot` expressions.

Definition 5.1.7 (Observation Event)**.** We call *observation event* any event associated with the execution of an instruction involving a `snapshot` expression.

In contrast to our first approach which induces a total order of observations, we allow programmers to specify an ordering between observation events. To this end, the following definitions introduce the *observation-ordering relation* as a precise tool in the hand of the programmers. This relation on observation events is derived from def-use, reference-based and in-memory data-flow relations, and control dependences.

Definition 5.1.8 (Dependence Relation)**.** We define relations $\overset{\text{du}}{\rightarrow}$, $\overset{\text{rf}}{\rightarrow}$ and $\overset{\text{cd}}{\rightarrow}$ as partial orders on def-use pairs, in-reference/in-memory data flow and control dependences, respectively. Formally, let $def(v, i)$ and $use(v, i)$ be the predicates evaluating to true if and only if instruction i defines variable v and instruction i uses variable v, respectively, and let *postdom* denote the post-domination (Cytron et al., 1991) binary predicate:

$$
\begin{aligned}
e_1 \overset{\text{du}}{\rightarrow}{}^{1}\, e_2 \quad & \text{if and only if} \quad def(v, Inst(e_1)) \wedge use(v, Inst(e_2)) \\
e_1 \overset{\text{rf}}{\rightarrow}{}^{1}\, e_2 \quad & \text{if and only if} \\
& \big(Inst(e_1) = (ref \texttt{ <- } v) \wedge Inst(e_2) = (var \texttt{ = } ref) \wedge \\
& \quad \nexists e_s,\ E = \ldots e_1 \ldots e_s \ldots e_2 \ldots,\ Inst(e_s) = (ref \texttt{ <- } v')\big) \\
& \vee \quad \big(Inst(e_1) = (\texttt{mem[}addr\texttt{] <- } v) \wedge Inst(e_2) = (var \texttt{ = mem[}addr\texttt{]}) \wedge \\
& \quad \nexists e_s,\ E = \ldots e_1 \ldots e_s \ldots e_2 \ldots,\ Inst(e_s) = (\texttt{mem[}addr\texttt{] <- } v')\big) \\
e_1 \overset{\text{cd}}{\rightarrow}{}^{1}\, e_2 \quad & \text{if and only if} \quad \exists e_s,\ E = \ldots e_1 \ldots e_s \ldots e_2 \ldots, \\
& \quad postdom(Inst(e_2), Inst(e_s)) \wedge \neg postdom(Inst(e_2), Inst(e_1))
\end{aligned}
$$

and

$$\overset{\text{du}}{\rightarrow} = \left(\overset{\text{du}}{\rightarrow}{}^{1}\right)^{*} \qquad \overset{\text{rf}}{\rightarrow} = \left(\overset{\text{rf}}{\rightarrow}{}^{1}\right)^{*} \qquad \overset{\text{cd}}{\rightarrow} = \left(\overset{\text{cd}}{\rightarrow}{}^{1}\right)^{*}$$

The dependence relation, denoted by $\overset{\text{dep}}{\rightarrow}$, is defined as the union of the def-use, reference-based and in-memory data-flow, and control dependence relations:

$$\overset{\text{dep}}{\rightarrow}_1 = \overset{\text{du}}{\rightarrow}_1 \cup \overset{\text{rf}}{\rightarrow}_1 \cup \overset{\text{cd}}{\rightarrow}_1 \qquad \text{and} \qquad \overset{\text{dep}}{\rightarrow} = \left(\overset{\text{dep}}{\rightarrow}_1 \right)^*$$

Definition 5.1.9 (Observe-From Relation)**.** Given an execution E, observation events induce a relation called *observe-from* and denoted by $\overset{\text{of}}{\rightarrow}$, mapping a definition to an observation event e_{obs}:

$$\forall \ldots e_1 \ldots e_{obs} \ldots \in E,\ \left(\mathit{Inst}(e_{obs}) = (\_ \texttt{ = snapshot}(\_)) \wedge e_1 \overset{\text{du}}{\rightarrow}_1 e_{obs} \right) \implies e_1 \overset{\text{of}}{\rightarrow} e_{obs}$$

Definition 5.1.10 (Observation Ordering Relation)**.** Any pair of distinct events associated with the execution of instructions involving `snapshot` expressions related through a dependence relation are ordered by a so-called *observation ordering* relation denoted by $\overset{\text{oo}}{\rightarrow}$. Formally, given an execution E of P, $\overset{\text{oo}}{\rightarrow}$ is the restriction of $\overset{\text{dep}}{\rightarrow}$ to observation events:

$$\forall \ldots e_1 \ldots e_2 \ldots \in E,\ \left(\mathit{Inst}(e_1) = (\_ \texttt{ = snapshot}(\_)) \wedge \mathit{Inst}(e_2) = (\_ \texttt{ = snapshot}(\_)) \wedge e_1 \overset{\text{dep}}{\rightarrow} e_2 \right) \implies e_1 \overset{\text{oo}}{\rightarrow} e_2$$

We chose to only include data flow relations (through SSA values, references or memory) and control dependences into $\overset{\text{oo}}{\rightarrow}$. This is a trade-off between providing more means to the programmer to constrain program transformations to enforce observation ordering, and freedom of code optimization left to the compiler in presence of such observations. Data-flow paths between `snapshot` expressions enable the expression of arbitrary partial orders of observation events, and they are easily under the control of programmers—if necessary by inserting dummy or token values as we will see in the next section—hence they appear to be expressive enough for our purpose. Note that control dependences are not directly useful at capturing partial ordering (that would not be otherwise expressible using data dependences), and it is sufficient to *not* make them dependent on the result of `snapshot` expressions to avoid having to unduly constrain the ordering of observations (e.g., forbidding legitimate hoisting of loop-invariant expressions). On the other hand, control dependences are important to model the effect of program transformations converting data dependences into control dependences: for example, boolean logic may be converted into control flow, yielding a single static truth value for some boolean variables occurring in a dependence chain linking two observations. Conversely, adding more relations into $\overset{\text{oo}}{\rightarrow}$ such as non-data-flow write-after-write (memory-based) dependences would not enhance the ability to represent more partial orders while severely restricting the compiler's ability to reorder loop iterations or hoist observations from loops.

Now that we have presented the operational and observation semantics of our Mini IR, as well as different relations involving observations, we need to define the notion of observation preservation for a program transformation.

5.1.4 Program Transformations

Let us first define a notion of program transformation, as general as possible, and without considering validity (correctness) issues for the moment. This notion is inseparable from a mapping that relates semantically connected events across program transformations.

Definition 5.1.11 (Program Transformation)**.** Given a program P, a transformation τ maps P to a transformed program P'. Every transformation τ induces an *event map* $\propto_\tau$ relating some events before and after transformation. The event map notation $e \propto_\tau e'$ reads as "τ maps e to e'", or "e maps to e'" when τ is clear from the context, or "τ preserves e" when the event after transformation does not need to be identified. The mapping is partial and neither injective nor surjective in general, as events in P may not have a semantically relevant counterpart in P' and vice versa.

In the following, we will incrementally construct a $\propto_\tau$ relation for an arbitrary transformation τ. Being a constructive definition, it will serve as a tool to prove the existence, ordering, and properties of values across program transformations. First, let us define the notion of *valid transformation*.

Definition 5.1.12 (Valid Program Transformation)**.** Given a program P, a program transformation τ that applies to P is valid if it produces a program $P' = \tau(P)$ such that $\forall I \in Inputs, \mathcal{C}[\![P]\!](I) = \mathcal{C}[\![P']\!](I)$, i.e. P and P' have the same I/O behavior.

The set of all valid transformations of P is denoted by $\mathcal{T}(P)$.

The set of hypotheses on what is considered a valid program transformation is minimal, covering as many compilation scenarios as possible. This constitutes a major strength of our proposal: we make no assumptions on the analysis and transformation power of a compiler, covering not only the classical scalar, loop and interprocedural transformations (optimization, canonicalization, lowering), but also hybrid static-dynamic schemes, including control and value speculation. The only constraint on transformations is to preserve the I/O behavior of a program *on all possible inputs*.

Let us now prove that I/O events as well as their relative ordering are preserved by all valid program transformations. We first introduce a class of events that are always related through $\propto_\tau$ for any valid transformation τ, then prove I/O events belong to this class.

Definition 5.1.13 (Transformation-Preserved Event)**.** Given a program P and input I, an event e_{tp} is *transformation-preserved* for execution $\mathcal{E}[\![P]\!](I)$ if all valid program transformations are guaranteed to preserve it.

Formally, $e_{tp} \in \mathcal{E}[\![P]\!](I)$ is a transformation-preserved event if and only if

$$\forall \tau \in \mathcal{T}(P),\ \exists e'_{tp} \in \mathcal{E}[\![\tau(P)]\!](I), e_{tp} \propto_\tau e'_{tp}$$

The set of transformation-preserved events for a program P and input I is denoted by $TP(P, I)$.

Let us now show that one may construct a $\propto_\tau$ relation that preserves I/O events.

Lemma 5.1.1 (Unicity of Transformed I/O Events)**.** *For an execution $E = \mathcal{E}[\![P]\!](I)$ of a program P on some input I, an event e from E reading or writing a value v from/to an input/output set, and a valid program transformation τ, there exists a unique event $e' \in \mathcal{E}[\![P']\!](I)$ such that e' reads or writes v.*

Proof. By Definition 5.1.12 (Valid Program Transformation), v also belongs to an input or output set associated with the transformed program $P' = \tau(P)$. As a consequence, $E' = \mathcal{E}[\![P']\!](I)$ also holds an event e' reading or writing v. Since v is uniquely tagged among I/O values, semantical equality $\mathcal{C}[\![P]\!](I) = \mathcal{C}[\![P']\!](I)$ implies that e' is the only event reading or writing v in the execution E'. □

Definition 5.1.14 (Preservation of I/O Events)**.** For an execution $E = \mathcal{E}[\![P]\!](I)$ of a program P on some input I and a valid program transformation τ, we define $\propto_\tau$ to include all pairs (e, e') such that e is an I/O event in E reading or writing a value *val* from/to an input/output set, and e' is the unique I/O event in $\mathcal{E}[\![P']\!](I)$ such that e' reads or writes *val*.

Lemma 5.1.2 (Preservation of I/O Event Ordering)**.** *Any valid program transformation preserves the partial ordering on I/O events.*

Proof. Consider the execution E of a program P on some input I, and a valid program transformation τ. Given two events e_1 and e_2 in E, each of which is associated with an `io` expression such that $e_1 \overset{\text{io}}{\rightarrow} e_2$. From Definition 5.1.14 (Preservation of I/O Events), there exists two events e'_1 and e'_2 in $E' = \mathcal{E}[\![\tau(P)]\!](I)$ such that $e_1 \propto_\tau e'_1$ and $e_2 \propto_\tau e'_2$. By definition of $\overset{\text{io}}{\rightarrow}$ induced by I/O events on input and output sets, any values $v_1 \in IO(e_1)$ and $v_2 \in IO(e_2)$ are such that $v_1 \overset{\text{io}}{\rightarrow} v_2$. Since τ is a valid transformation, events e'_1 and e'_2 also have to be ordered such that $v_1 \overset{\text{io}}{\rightarrow} v_2$, hence $e'_1 \overset{\text{io}}{\rightarrow} e'_2$. □

Finally, one may lift the notion of transformation preservation to a program instruction, collecting events associated with all or a subset of the executions of this instruction.

Definition 5.1.15 (Transformation-Preserved Instruction)**.** Given a program P, i_{tp} is a *transformation-preserved instruction* of P if all valid program transformations are guaranteed to preserve its associated events, for all inputs.

Formally, i_{tp} is a transformation-preserved instruction if and only if

$$\forall \tau \in \mathcal{T}(P), \forall I \in Inputs,\ \forall e_{tp} \in \mathcal{E}[\![P]\!](I),\ i_{tp} = Inst(e_{tp}),\ \exists e'_{tp} \in \mathcal{E}[\![\tau(P)]\!](I), e_{tp} \propto_\tau e'_{tp}$$

And i_{ctp} is *transformation-preserved conditionally on the preservation of an instruction* i_c if for all executions of P, the preservation of some event e associated with the execution of i implies the preservation of any event e_{ctp} associated with the execution of i_{ctp}.

Formally, i_{ctp} is conditionally transformation-preserved on i_c if and only if

$$\begin{aligned}&\forall \tau \in \mathcal{T}(P), \forall I \in Inputs,\ \forall e_{ctp} \in \mathcal{E}[\![P]\!](I),\ i_{ctp} = Inst(e_{ctp}),\\ &\exists \ldots e_{ctp} \ldots e_c \ldots \in \mathcal{E}[\![P]\!](I),\ i_c = Inst(e_c),\ \exists e'_c \in \mathcal{E}[\![\tau(P)]\!](I), e_c \propto_\tau e'_c\\ &\qquad\qquad \implies \exists \ldots e'_{ctp} \ldots e'_c \ldots \in \mathcal{E}[\![\tau(P)]\!](I), e_{ctp} \propto_\tau e'_{ctp}\end{aligned}$$

As will be shown in Section 5.2.3, we will use these notions to validate the preservation of security protections, either unconditionally, or conditionally on the execution of a given key instruction of the protection scheme.

5.1.5 Happens-Before Relation

Given the definition of program transformation established in the last subsection, we may now define the notion of observation-preserving transformation. Intuitively, such a transformation has to preserve observation events as well as the associated observed values, and furthermore preserve the relations involving observations such as $\overset{\text{of}}{\rightarrow}$ and $\overset{\text{oo}}{\rightarrow}$. To this end, let us first define a partial order on both I/O and observation events, capturing not only the I/O semantics of the program but also its associated observations.

Definition 5.1.16 (Happens-Before Relation)**.** For a given program execution E, one may define a partial order $\overset{\text{hb}}{\rightarrow}$ over pairs of events called a *happens-before relation*. It has to be a sub-order of the total order of events in E.

Definition 5.1.17 (Preservation of Happens-Before)**.** Given a valid program transformation τ, for any input $I \in \mathit{Inputs}$, P produces an execution $E = \mathcal{E}[\![P]\!](I)$, and the transformed program $P' = \tau(P)$ produces an execution $E' = \mathcal{E}[\![P']\!](I)$. τ is said to preserve the happens-before relation if any events in happens-before relation in P have their counterparts through $\propto_\tau$ in happens-before relation in P'. Formally,

$$\forall e_i, e_j \in E,\ \forall e'_i, e'_j \in E',\ e_i \overset{\text{hb}}{\rightarrow} e_j \ \wedge\ e_i \propto_\tau e'_i \ \wedge\ e_j \propto_\tau e'_j \implies e'_i \overset{\text{hb}}{\rightarrow} e'_j$$

Definition 5.1.17 (Preservation of Happens-Before) defines the notion of happens-before relation preservation, which is a property that has to be proven in general. Depending on how sparse the $\propto_\tau$ and $\overset{\text{hb}}{\rightarrow}$ relations are, it may be more or less difficult to enforce and establish. Hereinafter, we use the following happens-before relation:

$$\overset{\text{hb}}{\rightarrow} = \left(\overset{\text{io}}{\rightarrow} \cup \overset{\text{of}}{\rightarrow} \cup \overset{\text{oo}}{\rightarrow} \right)^*$$

Thanks to Lemma 5.1.2 (Preservation of I/O Event Ordering), one will only have to prove the preservation of the $\overset{\text{of}}{\rightarrow}$ and $\overset{\text{oo}}{\rightarrow}$ components of $\overset{\text{hb}}{\rightarrow}$ in the following. On the contrary, unlike I/O instructions, instructions involving `snapshot` are *not* preserved by valid program transformations in general.

We now provide two important definitions to reason about the preservation of observations.

Definition 5.1.18 (Observation-Preserving Transformation)**.** Given a program P, a transformation τ that applies to P is *observation-preserving* if it produces a program $P' = \tau(P)$ such that the four following conditions hold:

(i) it is a valid transformation (see Definition 5.1.12 (Valid Program Transformation));

(ii) it preserves the existence of observation events:

$$\begin{aligned}\forall I \in \mathit{Inputs}, \forall e \in \mathcal{E}[\![P]\!](I),\ \mathit{Inst}(e) = (\_ \texttt{ = snapshot(\_)}) \\ \implies \exists e' \in \mathcal{E}[\![P']\!](I),\ \mathit{Inst}(e') = (\_ \texttt{ = snapshot(\_)}) \ \wedge\ e \propto_\tau e'\end{aligned}$$

(iii) it preserves the observed values:

$$\begin{aligned}\forall I \in \mathit{Inputs}, \forall e \in \mathcal{E}[\![P]\!](I),\ \mathit{Inst}(e) = (\_ \texttt{ = snapshot(\_)}), \\ e' \in \mathcal{E}[\![P']\!](I),\ \mathit{Inst}(e') = (\_ \texttt{ = snapshot(\_)}) \ \wedge\ e \propto_\tau e' \\ \implies \mathit{Obs}(e) = \mathit{Obs}(e')\end{aligned}$$

(iv) it preserves all happens-before relations:

$$\forall I \in \mathit{Inputs},\ \forall e_1, e_2 \in \mathcal{E}[\![P]\!](I),\ e_1 \overset{\text{hb}}{\rightarrow} e_2 \implies \exists e'_1, e'_2 \in \mathcal{E}[\![P']\!](I),\ e'_1 \overset{\text{hb}}{\rightarrow} e'_2$$

Given a program P, a transformation τ that applies to P is *observation-preserving conditionally on* instruction i_c in P if it produces a program $P' = \tau(P)$ such that the conditions (i), (iii) and (iv) above hold, and also:

(ii$_c$) it preserves the existence of observation events conditionally on the preservation of i_c:

$$\begin{aligned}\forall I \in \mathit{Inputs}, \forall e \in \mathcal{E}[\![P]\!](I),\ \mathit{Inst}(e) = (\_ \texttt{ = snapshot(\_)}),\\ \exists e_c \in \mathcal{E}[\![P]\!](I),\ i_c = \mathit{Inst}(e_c),\ \exists e'_c \in \mathcal{E}[\![P']\!](I),\ e_c \propto_\tau e'_c\\ \implies \exists e' \in \mathcal{E}[\![P']\!](I),\ e \propto_\tau e'\end{aligned}$$

Note that a snapshot instruction is always transformed into a snapshot instruction observing the same observation state, by an observation-preserving transformation.

Let us now define a notion of observation that is preserved over all possible valid transformations.

Definition 5.1.19 (Protected Observation)**.** An observation in a program P is *protected* if and only if all valid transformations that apply to P are observation-preserving.

An observation is *protected conditionally on instruction* i_c if and only if all valid transformations are observation-preserving conditionally on i_c.

Note that the composition of two valid transformations yields a valid transformation, according to Definition 5.1.12 (Valid Program Transformation). As a consequence, Definition 5.1.19 (Protected Observation) covers compositions of valid transformations along a compilation pass pipeline.

In the next section, we will provide a constructive method to implement programs with *protected observations* complying with Definition 5.1.19 (Protected Observation). This will allow us to prove the preservation of $\overset{\text{hb}}{\to}$ on a class of programs with special constructs carefully defined to protect `snapshot` expressions, as a partial fulfillment of the requirements for a valid program transformation to be observation-preserving.

5.2 An Approach for Preserving Observations

As noted earlier, valid transformations do not preserve the happens-before relation in general. This section introduces a mechanism to achieve this, involving a minor extension of the Mini IR with expressions that are defined to be opaque to any program analysis. As explained in Section 4.2, the intuition of this opacification technique is to hide a given value from compiler optimizations, so that they cannot reason about it and thus cannot optimize it out. The technique has been illustrated through the introduction of artificial definitions. In this section, we will formalize the notion of opacification and prove that it indeed allows for protected observations.

5.2.1 Mini IR Extension: Opaque Expressions

To implement the preservation of observation events and the associated happens-before relation, we extend Mini IR with an *opaque expression* syntax. The `opaque` keyword introduces a region of control flow, as shown in Figure 5.2. The opaque expression syntax gets its name from the "opacity" of its enclosed region w.r.t. program analyses and transformations. An instruction defining a value from an opaque expression is called an *opaque instruction*.

When executing `opaque`, the associated event gathers the definitions and effects of all instructions in its enclosed region, *atomically and in isolation*. We assume the

extended-expr	::=	*expr*	
	|	`opaque` *region*	atomic opaque region: make I/O, side-effects and definitions visible atomically outside the region and vice versa; the compiler sees statically unknown yet functionally deterministic values
	|	`yield(` *var** `)`	return from atomic opaque region

FIGURE 5.2: Extension of Mini IR to implement event and happens-before preservation.

enclosed region is a *terminating* sequence of instructions. It proceeds with "internal" state transitions without defining events and without exposing intermediate states. When reaching a `yield` instruction, the program state serves as the resulting state of the atomic event while also defining all values listed in the `yield` instruction. Formally, executing an opaque instruction o enclosing a region $\{i_1\,;\ \ldots\,;\ i_n\}$ on a state σ yields the event $e = \sigma \overset{\mathrm{o}}{\rightsquigarrow} \sigma'$ where $\sigma \overset{\mathrm{i_1}}{\rightsquigarrow} \cdots \overset{\mathrm{i_n}}{\rightsquigarrow} \sigma'$. The last instruction i_n must be a `yield` instruction. We authorize arbitrary (terminating) control flow in these regions, including conditional memory access and I/O. As a result, the memory and I/O effects triggered by an opaque instruction are input-dependent. Since regions inside opaque expressions always terminate, σ' always exists. This definition guarantees both atomicity and isolation, since states and transitions associated with individual instructions $\{i_k\}_{1 \leq k \leq n}$ are not modeled in the operational semantics. Finally, the semantics of nested opaque expressions is defined inductively from the inside out.

The compiler is very limited in what analyses it may perform on opaque expressions:

- gathering the uses of an opaque expression;
- deciding whether an opaque expression has read or write side-effects;
- deciding whether an opaque expression performs I/O;
- deciding whether two opaque expressions are identical up to variable renaming.

Yet the compiler is not allowed to determine the precise side-effects (references, memory addresses) in an opaque expression (as will be shown in Definition 5.2.2 (Opaque Value Set), we will refer to the notion of *identical instruction*, and it would be much more complicated to reason about this notion if the compiler has more analysis capacity on opaque expressions), and it may not attempt to establish a correlation between its uses (resp. loads from references or memory) and the values it defines (resp. stores to references or memory).

On the other hand, a valid transformation τ may delete or duplicate an opaque instruction, and even synthesize completely new ones. This may sound too powerful, as without additional care τ may break the opacity and expose intermediate states in an opaque expression. We will see in the following that the opacity property itself prevents this from happening, thus maintaining opacity, atomicity and isolation of opaque expressions across transformations.

Since opaque expressions can nest multiple instructions (and even nested regions), we introduce a notation to denote the sequence of instructions executed atomically within an event. Let $\mathit{InstList}(e)$ denote the list of instructions associated with event e; it is a single-element list for all events, except for those associated

with opaque instructions where it is the sequence of instructions executing within the region for this particular instance e of the opaque instruction. We will write $i \in \mathit{InstList}(e)$ to denote that an instruction i is associated with event e.

In the rest of the paper, we will use revised and extended versions of Definitions 5.1.5–5.1.18 operating on *sets of instructions* in $\mathit{InstList}(e)$ rather than a specific instruction $\mathit{Inst}(e)$. All equalities of the form $i = \mathit{Inst}(e)$ in these equations should be rewritten into $i \in \mathit{InstList}(e)$. For convenience, we will also consider all I/O expressions as being opaque; this is consistent with the traditional assumptions about compilers not being able to analyze across system calls.

Informally, opaque expressions have two important consequences on value and event preservation across program transformations: (1) if a valid program transformation preserves an event using a value defined by an opaque instruction or stored by an instruction from its associated region, then it must also preserve the event associated with the opaque instruction, and (2) a valid program transformation has to preserve any value used in the opaque expression, as proceeding with downstream computation would otherwise involve some form of unauthorized guessing of the opaque expression's behavior. Formally, let the predicate $\mathit{Opaque}(e)$ denote that event e is associated with the execution of an opaque instruction; we restrict the effects of program transformations in presence of opaque expressions as follows:

Definition 5.2.1 (Opaque Expression Preservation)**.** Given a program P, an input I, and valid transformation τ,

$$\begin{aligned}\forall \ldots e_1 \ldots e_2 \ldots \in \mathcal{E}[\![P]\!](I),\ \mathit{Opaque}(e_1),\ e_1 \overset{\text{dep}_1}{\to} e_2,\ \exists e_2' \in \mathcal{E}[\![\tau(P)]\!](I),\ e_2 \propto_\tau e_2' \\ \implies \exists e_1' \in \mathcal{E}[\![\tau(P)]\!](I),\ e_1 \propto_\tau e_1' \ \wedge\ \mathit{Opaque}(e_1') \ \wedge\ e_1' \overset{\text{dep}}{\to} e_2' \end{aligned} \tag{5.1}$$

(Preservation of Opaque Expression)

$$\begin{aligned}\forall \ldots \sigma_e e \ldots \in \mathcal{E}[\![P]\!](I),\ \forall \ldots \sigma_e' e' \ldots \in \mathcal{E}[\![\tau(P)]\!](I), \\ \mathit{Opaque}(e),\ e \propto_\tau e',\ \mathit{use}(v, \mathit{Inst}(e)) \ \wedge\ (v, \mathit{val}) \in \sigma_e \\ \implies \exists i' \in \mathit{InstList}(e'), \mathit{use}(v', i') \ \wedge\ (v', \mathit{val}) \in \sigma_e' \end{aligned} \tag{5.2}$$

(Preservation of Value Used in Opaque Expression)

This restriction is taken as a definition, formally capturing the intuitive expectations about what the compiler has to enforce in the presence of opaque expressions.

Notice the transitive dependence relation $e_1' \overset{\text{dep}}{\to} e_2'$ in the transformed program (rather than $e_1' \overset{\text{dep}_1}{\to} e_2'$): the immediate dependence may be transformed into a series of instructions (e.g., spilling a value to the stack).

Let us highlight a subtle point in this definition: i' is an instruction belonging to the transformed opaque expression's region, not the opaque instruction itself. Indeed, variable v' may not be a free variable in $\mathit{Inst}(e')$, it may be bound to the opaque expression's internal region. For example, it is always correct to transform `opaque { use_of(v)}` into the sequence `t1 = not v; opaque { v'= not t1; use_of(v')}`. This does not involve any analysis—which is explicitly disallowed—of the opaque expression's semantics. While `v'` remains part of the program state and retains the value `v` had in the original program, it is not exposed as a variable captured by the opaque expression.

In the following, we will only use `snapshot` within the region of an opaque expression. As a result, `snapshot` expressions will inherit all properties of opaque expressions, including the conditions for their preservation (Property Preservation of

Opaque Expression) and the preservation of observed values (Property Preservation of Value Used in Opaque Expression).

5.2.2 Opaque Chains

Let us now build dependence chains involving opaque instructions. These will be called *opaque chains* and serve two purposes: (1) establishing a transformation-preserved $\overset{oo}{\rightarrow}$ relation, and (2) linking observations to downstream transformation-preserving events to preserve the former through program transformations. We first need additional definitions and notations.

Definition 5.2.2 (Opaque Value Set)**.** Given an execution $E = \mathcal{E}[\![P]\!](I)$, consider a chain of dependent events $e_1 \overset{dep_1}{\rightarrow} \dots \overset{dep_1}{\rightarrow} e_n$ with $n \geq 2$, an instruction/event $i_j = Inst(e_j)$, such that $Opaque(e_j)$, in the chain defining a value *orig* for a given variable var_j and some instruction/event $i_k = Inst(e_k)$ in the chain, with $1 \leq j < k \leq n$ and $\forall j < l < k, \neg Opaque(e_l)$. Notice that we have $e_1 \overset{dep}{\rightarrow} e_k$ and not necessarily $e_1 \overset{dep_1}{\rightarrow} e_k$, which means that *orig* is not necessarily used or read by e_k.

Let $\sigma_j = (Vals_j, \pi_j)$ be the program state e_j transitions into.

Let OV_j denote the set of all opaque values that var_j may take according to its data type: for example, an opaque expression yielding a value of boolean type will have $OV_j = \{\text{true}, \text{false}\}$.

We also lift this definition to the set of values used by a downstream expression across a chain of dependent instructions. This expresses the sensitivity of an expression to a value produced by an upstream instruction in the dependent chain.

Consider an execution E_{alt} continuing after e_j on program state $\sigma_{alt} = (Vals_{alt}, \pi_j)$, with the value set $Vals_{alt}$ containing an alternative value *alt* substituted for var_j, i.e. $Vals_{alt} = Vals_j \setminus (var_j, orig) \cup (var_j, alt)$, and an event $e_{k_{alt}} \in E_{alt}$ such that $e_j \overset{dep}{\rightarrow} e_{k_{alt}}$. Let $value_{j,k}$ denote the function mapping every value $v \in OV_j$ to a value defined as follows:

- if $v = orig$, $value_{j,k}(v)$ is the value used or read by i_k;
- otherwise, if $v \neq orig$:
  - $value_{j,k}(v)$ is the value used or read by i_k if $e_k \in E_{alt}$;
  - $value_{j,k}(v)$ is the value used or read by an instruction $i_{k_{alt}} = Inst(e_{k_{alt}})$ along the $e_j \dots e_{k_{alt}}$ sub-chain if i_k and $i_{k_{alt}}$ are identical expressions up to variable renaming, and $\forall e \neq e_{k_{alt}}$ s.t. $e_j \overset{dep}{\rightarrow} e \overset{dep}{\rightarrow} e_{k_{alt}}$, $\neg Opaque(e)$;
  - $value_{j,k}(v)$ is the special "undefined" value $\perp$ otherwise.

We define the opaque value set $OV_{j,k}$ as $value_{j,k}(OV_j)$.

Let us paraphrase this definition to expose the intuitions behind it. When considering the *orig* value, the $value_{j,k}$ function yields the value used or read by i_k. When substituting the value *alt* for the original value *orig* defined by the opaque expression i_j, the $value_{j,k}$ function also yields the value used or read by i_k if the execution path is not altered or if the altered path still reaches i_k. If the altered path (when considering the *alt* value) encounters an instruction $i_{k_{alt}}$ identical to i_k up to variable renaming before encountering a dependent opaque instruction, it yields the value used or read by $i_{k_{alt}}$ (this accounts for program transformations capable of combining two identical instructions, more on this later). And if the altered path reaches an

opaque instruction, or the execution terminates before reaching an identical instruction, $value_{j,k}$ yields the "undefined" value $\perp$.

Intuitively, the definition of $value_{j,k}$ allows to reason on the cardinality of the opaque value set $OV_{j,k}$: whether this set is a singleton or not will tell whether the dependent instruction/event $i_k = Inst(e_k)$ is truly sensitive to the opaque value of $i_j = Inst(e_j)$. A non-singleton set means that evaluating the opaque instruction i_j cannot be avoided by a valid transformation, implying that the latter has to preserve i_j in order to preserve the resulting opaque value.

This notion of opaque value set lays the foundation for the definition of opaque chain below, which will be later needed to preserve the $\overset{\text{dep}}{\rightarrow}$ relations between opaque expressions. Moreover, we will further illustrate this concept of opaque value set on a handful of examples.

Definition 5.2.3 (Opaque Chain)**.** Given an execution $E = \mathcal{E}[\![P]\!](i)$, consider a chain of dependent events $e_1 \overset{\text{dep}_1}{\rightarrow} \ldots \overset{\text{dep}_1}{\rightarrow} e_n$; $e_1 \overset{\text{dep}_1}{\rightarrow} \ldots \overset{\text{dep}_1}{\rightarrow} e_n$ is an *opaque chain* linking e_1 to e_n if and only if

(i) $i_1 = Inst(e_1)$ and $i_n = Inst(e_n)$ are opaque instructions;

(ii) for any opaque instruction $i_k = Inst(e_k)$ other than i_1 ($2 \leq k \leq n$), let $i_j = Inst(e_j)$ its immediately preceding opaque instruction in the chain ($1 \leq j < k$), $\text{Card}(OV_{j,k}) \geq 2$.

We note $e_1 \overset{\text{opaque}}{\rightsquigarrow} e_n$ such an opaque chain.

As a corollary, any sub-chain of an opaque chain starting and ending with opaque instructions is trivially an opaque chain itself.

The intuition behind the condition (ii) is the following. In order to not break the opacity of the chain, the compiler must not be able to reason about any value defined by every opaque instruction $i_{opaque} = Inst(e_{opaque})$ of the chain, so that it is enforced to preserve the associated opaque event e_{opaque}. To this end, for every opaque instruction i_k of the chain other than i_1, we consider all values that its immediately upstream opaque instruction in the chain i_j may define, according to its opaque result type. If the opaque value set $OV_{j,k}$ of possible values used or read by i_k (which is on a dependence relation with the value defined by i_j, as $e_j \overset{\text{dep}}{\rightarrow} e_k$) contains at least two elements, then the compiler has no other way to compute this value used or read by i_k than using the opaque value defined by i_j. In other words, i_k is truly dependent on the opaque value of i_j, and the presence of the i_k actually requires the presence of i_j, given such dependence relation. The cardinality requirement of $OV_{j,k}$ implies that either i_k is control-dependent on i_j and there exists an alternate execution from e_j bypassing i_k (or an identical instruction up to variable renaming), or i_k is data-dependent on i_j and the set of values i_k may use or read is not a singleton, or both.

Opaque chains take the form of an alternating sequence of opaque instructions and sub-chains of regular instructions, starting with an opaque instruction and ending with an opaque instruction (remember I/O expressions are considered opaque). The control- and data-dependence restrictions in condition (ii) serve as "information-carrying" guarantees: the compiler does not have enough information about the possible paths dependent on an opaque value or on the processing of opaque values to break an opaque chain into distinct dependence chains.

Beyond opaque instructions, here are some important classes of instructions which always belong to an opaque chain:

- all instructions that only propagate existing values within or across name-value domains; these include dereference, assignment, load, store, `br` on branch arguments (not on the branch condition), `call` and `return` instructions, and instructions involving `snapshot` or `io` expressions;
- the same applies to the traditional C unary operators `-`, `!`, `~`;
- any binary operator (resp. function call) where the operand (resp. arguments) type or the value of the other operand (resp. other arguments) makes the operation bijective; e.g., `+` on unsigned integers, `*` with the constant 1, etc.
- any binary operator (resp. function call) where all operands (resp. arguments) are opaque, and where opaque operands (resp. arguments) are not correlated (feeding multiple times the same opaque value or dependent expressions may degenerate into a singleton value set, such as the subtraction of an opaque value with itself);

More instructions may belong to an opaque chain provided specific constraints hold on its inputs: e.g., left-shifting by 1 an `uint32_t` value if the compiler cannot prove it is always greater than or equal to `UINT_MAX/2`, or dividing a value that the compiler cannot statically analyze to be less than the divisor; in both cases the compiler is forced to consider that the image of the instruction on all possible inputs is not a singleton.

Example 5.2.1. Let us demonstrate this definition on a simple example of opaque chain. Consider the following code snippet, which simply outputs an input value incremented by 1 (assuming in the following, `io()` expression performs the I/O operation of printing an integer value to the terminal):

```
1 bb_entry:
2   integer_input = get_input();
3   a = opaque {
4     yield(integer_input);
5   };
6   b = a + 1;
7   io(desc, b);
```

Consider a dependence chain of length three, comprising the following instructions/events:

- $i_1 = Inst(e_1)$ is `a = opaque { yield(integer_input); };`;
- $i_2 = Inst(e_2)$ is `b = a + 1;`;
- $i_3 = Inst(e_3)$ is `io(desc, b);`.

Condition (i) holds because i_1 and i_3 are opaque instructions (recalling that I/O expressions are also opaque). To reason about the condition (ii), consider the only opaque instruction i_3 (other than i_1) of the chain, with i_1 being its immediately preceding opaque instruction. $e_1 \xrightarrow{\text{du}} e_3$ and the incrementation performed by i_2 is a bijective operation, thus does not modify the cardinality of the opaque value set OV_1 (i.e. the set of opaque values that a may take, which is 2^{32}, assuming an integer of 32 bits). Hence, $\text{Card}(OV_{1,3}) = \text{Card}(OV_1) = 2^{32}$, i.e. the value b used by i_3 is not a singleton.

```
bb_entry:
  integer_input = get_input();
  a = opaque {
    yield(integer_input);
  };
  b = a << 32;
  io(desc, b);
```

Example 5.2.2. On the contrary, considering a counter-example of opaque chain below:

The only change we have made here is that i_2 now becomes `b = a << 32;`. Assuming `integer_input` is of type `uint32_t`, shifting it by 32 bits to the left would allow the compiler to deduce that `b` always takes the zero value, i.e. $\text{Card}(OV_{1,3}) = 1$. As a result, the compiler may further eliminate both the opaque instruction i_1 and i_2, transforming the code into:

```
bb_entry:
io(desc, 0);
```

Indeed, the original dependence chain is not opaque, and valid transformations are not required to preserve the opaque instruction i_1, as there is no instruction really sensitive to the value produced by i_1.

Example 5.2.3. As a more complex example, the following code from illustrates the conversion of control into data dependences and vice-versa, preserving opaque chains in the process.

Consider Listing 5.1, both programs P_{data} and P_{control} form dependence chains from the definition of `c` to the observation of `v` (for P_{data}), or of 0 and 42 for P_{control}.

```
bb_entry:
  boolean_input = get_input();
  c = opaque {
    yield(boolean_input);
  };
  br c, bb_true;
bb_false:
  v = 0;


  br true, bb_join;
bb_true:
  v = 42;


bb_join:
  opaque {
    yield(snapshot(v));
  };
  ...
```

(a) P_{data}

```
bb_entry:
  boolean_input = get_input();
  c = opaque {
    yield(boolean_input);
  };
  br c, bb_true;
bb_false:
  opaque {
    yield(snapshot(0));
  };
  br true, bb_join;
bb_true:
  opaque {
    yield(snapshot(42));
  };
bb_join:




  ...
```

(b) P_{control}

Listing 5.1: Opaque chain preservation during data and control dependences conversion.

More specifically, for P_{data} from Listing 5.1a, we have two dependence chains C_{data} and C'_{data}, each of length three, containing respectively:

- $i_1 = Inst(e_1)$ is `c = opaque { yield(boolean_input); };;`
- $i_2 = Inst(e_2)$ is `v = 0;;`

- $i_3 = Inst(e_3)$ is `opaque { yield(v); };`.

and:

- $i'_1 = Inst(e'_1)$ is `c = opaque { yield(boolean_input); };;`
- $i'_2 = Inst(e'_2)$ is `v = 42;;`
- $i'_3 = Inst(e'_3)$ is `opaque { yield(v); };`.

Let us now reason about the opacity of C_{data}. Condition (i) holds trivially because i_1 and i_3 are opaque instructions. $\text{Card}(OV_1) = 2$ as c is of type `bool`, and $e_1 \overset{\text{cd}}{\rightarrow}^1 e_2$, which creates two different execution paths, each of which defines a distinct value for v (respectively 0 and 42). $e_2 \overset{\text{du}}{\rightarrow}^1 e_3$, making $\text{Card}(OV_{1,3}) = 2$, as the variable v used by i_3 also has two possible values. Following the same reasoning, C'_{data} is also an opaque chain.

Let us now consider P_{control} from Listing 5.1b, which also defines two dependence chains $C_{control}$ and $C'_{control}$, each of length two, containing respectively:

- $i_1 = Inst(e_1)$ is `c = opaque { yield(boolean_input); };;`
- $i_2 = Inst(e_2)$ is `opaque { yield(0); };`.

and:

- $i'_1 = Inst(e'_1)$ is `c = opaque { yield(boolean_input); };;`
- $i'_2 = Inst(e'_2)$ is `opaque { yield(42); };`.

Let us now reason about the opacity of $C_{control}$. Condition (i) holds trivially because i_1 and i_2 are opaque instructions. To reason about the condition (ii), we now have to determine the opaque value set $OV_{1,2}$. Consider the variable c of type `bool`, $\text{Card}(OV_1) = 2$, $e_1 \overset{\text{cd}}{\rightarrow}^1 e_2$. As i_2 lies in the `bb_false` block, *orig* takes the boolean value `false`, thus the only alternative value *alt* equals `true`.

- Considering $c = orig$, $value_{1,2}(c)$ yields the value 0 used by i_2;
- Substituting *alt* for *orig*, $value_{1,2}(c)$ yields the "undefined" value $\perp$, as i_2 and i'_2 (which is the first opaque instruction in the alternative execution path) are not identical: they consume two distinct values 0 and 42.

Therefore, $OV_{1,2} = \{0, \perp\}$, and $\text{Card}(OV_{1,2}) = 2$; $C_{control}$ is indeed an opaque chain. Following the same reasoning, for $C'_{control}$, $OV_{1,2} = \{42, \perp\}$, $\text{Card}(OV_{1,2}) = 2$, and $C'_{control}$ is also an opaque chain.

In short, the transformations from P_{data} to P_{control} and vice-versa are both valid, preserving observations (a data dependence is converted into a control dependence, specializing values into constants, and vice-versa for the reverse transformation). Whether it is the multiple values of `v` or the alternative path from the definition of `c` to a consuming `snapshot`, it is impossible for the compiler to break the dependence.

Example 5.2.4. On the contrary, let us illustrate the restrictions on the multiplicity of paths leading and not leading to i_k or an identical opaque expression. Consider Listing 5.2, where a program P_{orig} (Listing 5.2a) forming a dependence chain from the definition of `c` to the I/O instruction through a control dependence, and its transformation into P_{trans} (Listing 5.2b).

More specifically, P_{orig} defines two opaque chains C and C', each of length two, containing respectively:

```
1  bb_entry:
2    boolean_input = get_input();
3    c = opaque {
4      yield(boolean_input);
5    };
6    br c, bb_true;
7  bb_false:
8    io(desc, 0);
9    br true, bb_join;
10 bb_true:
11   io(desc, 0);
12 bb_join:
13   ...
```

(a) P_{orig}

```
1  bb_entry:
2    boolean_input = get_input();
3    c = opaque {
4      yield(boolean_input);
5    };
6
7
8
9
10
11
12   io(desc, 0);
13   ...
```

(b) P_{trans}

Listing 5.2: Non-opaque control dependence chain.

- $i_1 = Inst(e_1)$ is `c = opaque { yield(boolean_input); };;`
- $i_2 = Inst(e_2)$ is `io(desc, 0);` (from `bb_false`).

and:

- $i'_1 = Inst(e'_1)$ is `c = opaque { yield(boolean_input); };;`
- $i'_2 = Inst(e'_2)$ is `io(desc, 0);` (from `bb_true`).

Let us now reason about the opacity of C. Condition (i) holds trivially because i_1 and i_2 are opaque instructions. Similarly to the previous example, we have $\text{Card}(OV_1) = 2$, $e_1 \xrightarrow{\text{cd}}_1 e_2$, the original value *orig* of c takes the boolean value `false`, while the only alternative value *alt* equals `true`.

- Considering $c = orig$, $value_{1,2}(c)$ yields the value 0 used by i_2;
- However, this time, by substituting *alt* for *orig*, $value_{1,2}(c)$ also yields the value 0 used by i'_2, which is identical to i_2.

Therefore, $OV_{1,2} = \{0\}$, and $\text{Card}(OV_{1,2}) = 1$; C does not form an opaque chain. Following the same reasoning, C' is not an opaque chain either, as $\text{Card}(OV_{1,2}) = 1$. In other words, the dependent instruction i_2 is not truly sensitive on the opaque value defined by i_1).

Indeed, for example, by applying the "tail merging" transformation to P_{orig}, we will obtain P_{trans} which contains no dependence anymore. As a result, a further "dead code elimination" transformation applied to P_{trans} will remove the opaque instruction `c = opaque { yield(boolean_input); };`, and this is completely valid, as the control dependence chain in P_{orig} is not an opaque one due to the identical values (constant 0) read by the I/O instruction on both paths leading to an identical instruction.

We may now generalize Property Preservation of Opaque Expression to opaque chains. If a valid program transformation preserves an event at the tail of an opaque chain, then it must also preserve the event associated with the head of the opaque chain. Formally:

Lemma 5.2.1 (Preservation of Chained Opacity)**.** *Given a program P, input I, and valid transformation τ,*

$$\forall \dots e_1 \dots e_n \dots \in \mathcal{E}[\![P]\!](I),\ e_1 \overset{\text{opaque}}{\rightsquigarrow} e_n, \exists e'_{n'} \in \mathcal{E}[\![\tau(P)]\!](I), e_n \propto_\tau e'_{n'}$$
$$\implies \exists e'_1 \in \mathcal{E}[\![\tau(P)]\!](I),\ e_1 \propto_\tau e'_1 \ \wedge\ \mathit{Opaque}(e'_1) \ \wedge\ e'_1 \overset{\text{dep}}{\rightarrow} e'_{n'}$$

Proof. The $\overset{\text{opaque}}{\rightsquigarrow}$ relation implies e_1 and e_n are opaque. The case of $n = 1$ is trivial.

Consider the case of $n \geq 2$. Let $i_k = \mathit{Inst}(e_k)$ be the immediately upstream opaque instruction/event of e_n in the chain. Consider the $e_k \dots e_n$ sub-chain (which is trivially an opaque chain). By definition of the $\overset{\text{opaque}}{\rightsquigarrow}$ relation, $\text{Card}(OV_{k,n}) \geq 2$. In other words, the sub-chain (excluding e_k and e_n) implements the function $value_{k,n}$ and it is sensitive to the value d defined by e_k (i.e., non-constant along the values d may take). Since the instructions in our Mini IR capable of implementing a non-constant function are the definition, load, store and conditional branches, this implies the existence of a slice of events in $\mathcal{E}[\![P]\!](I)$, spawning backward from e_n and including an event e_u using d. It is trivial that $e_k \overset{\text{dep}_1}{\rightarrow} e_u$. The events associated with these definition, load, store, branch instructions are exactly those building up $\overset{\text{dep}}{\rightarrow}$. As a result, from Property Preservation of Value Used in Opaque Expression, e_n being opaque, the mapping of e_n to $e'_{n'}$ implies that the same value $value_{k,n}(d)$ is used by $e'_{n'}$. As a result, there exists in $\tau(P)$ a function $value'_{k',n'}$ sensitive to d, which is in turn computed by a slice in $\mathcal{E}[\![\tau(P)]\!](I)$ spawning backward from $e'_{n'}$. The sensitivity of $value'_{k',n'}$ to the value of d implies that the backward slice holds an event $e'_{u'}$, such that $e_u \propto_\tau e'_{u'}$, using this latter. We may apply Property Preservation of Opaque Expression to $e_k \overset{\text{dep}_1}{\rightarrow} e_u$ and $e_u \propto_\tau e'_{u'}$, which proves the existence of an opaque event $e'_{k'} \in \mathcal{E}[\![\tau(P)]\!](I)$ such that $e_k \propto_\tau e'_{k'}$ and $e'_{k'} \overset{\text{dep}}{\rightarrow} e'_{n'}$.

An induction on the length of the chain proves the preservation of e_1 through valid transformations for all chain lengths, and that the transformed events form a dependence chain $e'_1 \overset{\text{dep}}{\rightarrow} e'_{n'}$. □

Note that $e_1 \overset{\text{opaque}}{\rightsquigarrow} e_n$ does not necessarily imply $e'_1 \overset{\text{opaque}}{\rightsquigarrow} e'_{n'}$, but does imply $e'_1 \overset{\text{dep}}{\rightarrow} e'_{n'}$ instead. Indeed, as $\mathit{Opaque}(e'_1)$, $e'_1 \overset{\text{dep}}{\rightarrow} e'_{n'}$ (Lemma 5.2.1 (Preservation of Chained Opacity)), and $\mathit{Opaque}(e'_{n'})$ (because $e_n \propto_\tau e'_{n'}$), all we have to do to prove $e'_1 \overset{\text{opaque}}{\rightsquigarrow} e'_{n'}$ is the cardinality requirement. However, without details about other transformed instructions of the chain, we cannot conclude whether the cardinality requirement is met. Notice also that events associated with non-opaque expressions along the chain are not necessarily preserved, so a k^{th} event ($k > 1$) of the opaque chain in P may be turned into the event number k' of the dependence chain in $\tau(P)$.

As a corollary, let us now introduce an important theorem on the preservation of opaque chains.

Theorem 1 (Preservation of Opaque Chains)**.** *Given a program P and input I, if e_1 is linked through an opaque chain to a transformation-preserved event e_n, then e_1 is transformation-preserved, and for any transformation τ mapping e_1 to e'_1 and e_n to $e'_{n'}$, there is an opaque chain linking e'_1 to $e'_{n'}$. Formally,*

$$e_1 \overset{\text{opaque}}{\rightsquigarrow} e_n \ \wedge e_n \in TP(P, I)$$
$$\implies \forall \tau \in \mathcal{T}(P),\ \exists e'_1, e'_{n'} \in \mathcal{E}[\![\tau(P)]\!](I),\ e_1 \propto_\tau e'_1 \ \wedge\ e_n \propto_\tau e'_{n'} \ \wedge\ e'_1 \overset{\text{dep}}{\rightarrow} e'_{n'}$$

Notice that a typical case of transformation-preserved e_n is an I/O event, but the lemma is not limited to opaque chains terminating in a consuming I/O event. For example, an opaque chain can also terminate in a consuming instruction that is *transformation-preserved conditionally* on some other instruction i_c (more on this in Theorem 2).

Unfortunately, as noted earlier, we have not been able to prove that an opaque chain transforms into an opaque chain in general. Additional hypotheses on opaque expressions or opaque chains are likely to be needed to prove this. We conjecture these would not require modifying our definition of opaque chains, but we do not have a definite answer at this point. Fortunately, we do not need to prove such a strong preservation result: a weaker-yet-sufficient compositionality result can be used as a work-around.

Lemma 5.2.2 (Transitive Preservation of Opaque Chains)**.**

$$\begin{aligned} &e_1 \overset{\text{opaque}}{\rightsquigarrow} e_n \ \wedge\ e_n \in TP(P, I) \\ &\quad\implies \forall \tau \in \mathcal{T}(P), \forall \tau' \in \mathcal{T}(\tau(P)),\ \exists e_1'', e_{n'}'' \in \mathcal{E}[\![\tau'(\tau(P))]\!](I), \\ &\qquad\qquad e_1 \propto_{\tau' \circ \tau} e_1'' \ \wedge\ e_n \propto_{\tau' \circ \tau} e_{n'}'' \ \wedge\ e_1'' \overset{\text{dep}}{\rightarrow} e_{n'}'' \end{aligned}$$

Proof. The proof of transitive transformation preservation stems from the observation that $\tau \circ \tau'$ is a valid transformation and the application of Theorem 1. □

This weaker result may sound counter-intuitive at first. It seems like establishing an opaque chain and applying τ then τ' in sequence provides weaker observation preservation guarantees than applying $\tau \circ \tau'$ in one shot. This is not the case. What happens is that after applying τ, we lose the ability to prove the original opaque chains will remain enforceable in the transformed program, hence to tell anything about the effect of τ' on the associated observations. The fact is that any observation τ preserves will also be preserved by $\tau \circ \tau'$, or any valid transformation. In the following, we will use Lemma 5.2.2 (Transitive Preservation of Opaque Chains) to reason about the preservation of opaque events throughout the compilation flow by composing transformations resulting from a sequence of compilation passes, rather than considering the cumulative effects of individual compilation passes on opaque chains.

5.2.3 Observation in action

Previously, in Listing 5.1, we have hinted at embedding `snapshots` into the region of opaque expressions. However, this is only the most basic implementation of observations. Let us now show how to combining `io`, `snapshot` and `opaque` expressions to create different observation schemes with varying degrees of freedom for the instruction scheduler and optimizations in general. These schemes are implemented as *patterns* in our Mini IR, but can be generalized to allow programmers to specify observations as well as the $\overset{\text{hb}}{\rightarrow}$ order to be preserved by the compiler in source program. First, let us start by defining some helper patterns needed to implement observation patterns.

5.2.3.1 Helper Patterns

Let us now introduce a simple pattern implementing a "tokenizing" opaque expression.

```
macro token(v1, ..., vk) { // pure, opaque unit-type value,
                           // not associated with any resource;
                           // the compiler sees a statically unknown
                           // yet functionally deterministic value.
bb:
  w = opaque {
    use(v1, ..., vk);      // variadic function using all its arguments
                           // and returns no value.
    yield(unit_value);
  };
  return(w);
}
```

It is pure, functionally deterministic, and opaque to the compiler. Semantically, the `token` pattern returns a value of the unit type—called a token—irrespectively of the number of arguments. Yet it is not known to the compiler what values a token can take, and in particular the compiler is not told it is a singleton set. As a result one may use `token` as an opaque expression, and use the token type in dependent instructions forming an opaque chain. Unlike more general opaque expressions, token values do not need hardware resources to store live token values when emitting assembly code: we refer to the unique value `unit_value` of the unit-type to implement "resource-less" opaque chains of tokens.

It may sound paradoxical to return the same `unit_value` in multiple tokens, yet this is not visible to the compiler since it occurs in an opaque expression; the compiler must assume these are different, unpredictable values.

Furthermore, since the unit data type does not carry an informative value, we define `snapshot` to ignore its token arguments, i.e., not to embed them into an observation state.

Next, we introduce a `tailio` pattern implementing a special case of the `io` instruction with no argument. It is typically used at the tail of opaque chains to prevent them from being eliminated.

```
macro tailio() {
bb:
  io(tagged_unit_unordered_set_descriptor);
  return();
}
```

The `tailio` pattern uses a dedicated descriptor of an unordered set of (tagged) unit-valued outputs. Since it embeds an I/O instruction, it is preserved by transformations, yet unlike the more general form of I/O, it does not incur any ordering constraint w.r.t. other `tailios` in the same program.

5.2.3.2 Robust Observation

Based on the extended patterns, we may now define a series of observation patterns, defining observation states that will be preserved over program transformations. They are ordered by increasing degrees of freedom for optimizations.

Monolithic opaque expression: This is the simplest form of observation preserved over program transformations: it is basically an opaque region containing a `snapshot` instruction observing a variable number of values and an I/O instruction to guarantee the preservation of the *whole* region. The opaque region also returns a token, implying that the observation does not require any physical resource to store the return value.

```
macro observe_monolithic(v1, ..., vk) {
bb:
  t2 = opaque {
    w1, ..., wk = snapshot(v1, ..., vk);
    t1 = token(w1, ..., wk);
    tailio();
    yield(t1);
  };
  return(t2);
}
```

Lemma 5.2.3 (Preservation of Monolithic Observation Events)**.** *If (1) all instructions involving* `snapshot` *expressions of a program P are introduced via* `observe_monolithic`*, and if (2) the happens-before relation on observations in P is enforced through an opaque chain, then observations in P are protected according to Definition 5.1.19 (Protected Observation).*

Proof. Consider any valid transformation τ. We need to prove that τ preserves monolithic observation event, according to Definition 5.1.18 (Observation-Preserving Transformation).

Condition (i) trivially holds, as τ is a valid transformation.

The opaque expression's atomic region expands a nested `tailio` pattern. As a result, Definition 5.1.14 (Preservation of I/O Events) guarantees that all events associated with the execution of such an opaque expression are transformation-preserved. This proves condition (ii).

The opaque expression's region holds a backward slice linking atomically the (token) value `t1`—which is also the return value of the region—to the arguments `v1,...,vk` to be observed. By Property Preservation of Value Used in Opaque Expression, any valid transformation must preserve the values of `v1,...,vk` along with the definition events producing these values. This proves condition (iii) in Definition 5.1.18 (Observation-Preserving Transformation).

Let us now prove condition (iv)—the preservation of $\overset{\text{hh}}{\rightarrow}$. The preservation of $\overset{\text{of}}{\rightarrow}$ follows the same reasoning as the proof of condition (iii); yet the preservation of $\overset{\text{oo}}{\rightarrow}$ involves the traversal of opaque chains. Consider a pair of events e_{obs_1}, e_{obs_2}, each of which is associated with the opaque expression expanded from `observe_monolithic`, such that $e_{obs_1} \overset{\text{oo}}{\rightarrow} e_{obs_2}$, and let e'_{obs_1}, e'_{obs_2} be their counterparts in $P' = \tau(P)$; these transformed events must exist as they involve I/O effects. Since $e_{obs_1} \overset{\text{opaque}}{\rightsquigarrow} e_{obs_2}$ (from the lemma statement), Theorem 1 applies to all `opaque` events on the opaque chain, guaranteeing their mapping to events in P' through τ, and that these events form a dependence chain. This proves $e'_{obs_1} \overset{\text{oo}}{\rightarrow} e'_{obs_2}$. □

Notice that some arguments of `observe_monolithic` associated with e_{obs_2} may be converted by τ into constants, which are defined at the initial event e_0. Still, at least one argument will remain the target of the opaque chain linking e_{obs_1} to e_{obs_2}, since the token-typed value produced by an upstream observation cannot be turned into a constant.

Decoupled I/O region: Cutting out a specific I/O section allows to decouple event preservation from event ordering. Rather than inserting an I/O instruction in every observation, it is sufficient to consider a smaller non-empty set of I/O instructions, and to chain this set backwards to the observations whose events they are meant to preserve.

The programmer may use `observe_decoupled` pattern to organize observations, setting partial ordering constraints among them using the resulting token `t1`. These

```
macro observe_decoupled(v1, ..., vk) {
bb:
  u1 = opaque {
    w1, ..., wk = snapshot(v1, ..., vk);
    t1 = token(w1, ..., wk);
    yield(t1);
  };
  return(u1);
}

macro observe_tailio(u1, ..., uk) {
bb:
  v = opaque {
    u = token(u1, ..., uk);
    tailio();
    yield(u);
  };
  return(v);
}
```

will be much less intrusive to compiler transformations than `observe_monolithic` which systematically embeds an I/O instruction.

We can now adapt the Lemma 5.2.3 (Preservation of Monolithic Observation Events) to the decoupled observation pattern.

Lemma 5.2.4 (Preservation of Decoupled Observation Events)**.** *If (1) all `snapshot` instructions of a program P are introduced via `observe_decoupled`, if (2) the happens-before relation on observations in P is enforced through opaque chains whose tails are I/O events introduced via `observe_tailio`, then observations in P are protected according to Definition 5.1.19 (Protected Observation).*

Proof. Similarly to Lemma 5.2.3 (Preservation of Monolithic Observation Events), we now need to prove that any valid transformation τ preserves decoupled observation event, according to Definition 5.1.18 (Observation-Preserving Transformation).

Condition (i) trivially holds, as τ is a valid transformation.

Since `observe_tailio` includes a `tailio` instruction, Definition 5.1.14 (Preservation of I/O Events) guarantees that events associated with instances of the opaque expression holding the `tailio` instruction are transformation-preserved. Theorem 1 applied to opaque chains linking `observe_decoupled` opaque expressions to a subsequent `observe_tailio` proves condition (ii).

The proof of condition (iv)—the preservation of $\overset{\text{hb}}{\rightarrow}$—in Lemma 5.2.3 (Preservation of Monolithic Observation Events) applies to `observe_decoupled` as it does not refer to the `tailio` instruction.

The proof of condition (iii) is also identical to the proof of Lemma 5.2.3 (Preservation of Monolithic Observation Events). □

Note that, just as for `tailio`, `observe_tailio` typically occurs at the tail of an opaque chain. Furthermore, a single `observe_tailio` may be used to close multiple opaque chains, as it accepts a variable number of arguments.

5.2.3.3 Address-Value Pair Observation

When observing values in memory, several applications require observing not only the value but also the memory address that holds this value. For example, this is important when verifying and assessing the proper erasure of a buffer in memory, to avoid leaking sensitive data. The following pattern provides such a functionality, when associated with an `observe_tailio` pattern as in the decoupled scheme above.

```
macro observe_pair(a) {
bb_macro:
  u = opaque {
    v = mem[a];
    b, w = snapshot(a, v);
    t = token(b, w);
    yield(t);
  };
  return(u);
}
```

The observation of both the address `a` and the value stored at this address occurs atomically, including these to the same observation state. One may use `observe_pair` to check the value at a specific memory address, as required by the abovementioned memory erasure example.

Lemma 5.2.5 (Preservation of Address-Value Pair Observation Events)**.** *If (1) all `snapshot` instructions of a program P are introduced via `observe_pair`, if (2) the happens-before relation on observations in P is enforced through opaque chains whose tails are I/O events introduced via `observe_tailio`, then observations in P are protected according to Definition 5.1.19 (Protected Observation).*

The proof of Lemma 5.2.5 (Preservation of Address-Value Pair Observation Events) is exactly the same as for Lemma 5.2.4 (Preservation of Decoupled Observation Events), as the only difference between two patterns `observe_decoupled` and `observe_pair` is that for the latter, the opaque expression contains an additional memory load instruction.

Of course, one may also define a version of this pattern for the monolithic scheme, and versions with a variable number of address-value pairs.

5.2.3.4 I/O-Barrier-Based Observation

Let us now use our extended syntax to implement the I/O-barrier-based solution to functional property preservation, described in Section 4.2. Recalling that preserving functional property is defined as the preservation of (1) values occurring in the property predicates and (2) the program observation point at which the property is evaluated. As such, the I/O-barrier-based mechanism is divided in two parts: first, the observation point associated with a given functional property is materialized, then Algorithm 1 inserts artificial definitions for every definition reaching the functional property. The former can be implemented with the following `observe_cc` pattern, while the latter corresponds to the following `artificial_def` pattern.

For each program property, the `observe_cc` pattern is used to model the associated observation point, as it snapshots values of observed variables and observed memory locations referred in the property's predicate. Note that for each observed memory location (i.e. `a1`, ..., `ak`), the pattern reads the value stored at the location, which is next snapshotted. This implements the compiler fence representing observation points, described in Section 4.2.

For each observed variable referred in the property's predicate, the `artificial_def` pattern is used to protect the value reaching the observation point, implementing the so-called opacification mechanism by returning opaque values. Recalling that these opaque values next replace the original ones in the subsequent code.

Notice that both `artificial_def` and `observe_cc` used for property preservation contain an I/O instruction: this guarantees the existence of these expressions. Furthermore, unlike the lightweight `observe_decoupled` pattern, these I/O instructions

```
macro artificial_def_cc(v) {
bb_macro:
  u = opaque {
    io(ordered_set_descriptor);
    yield(v);
  };
  return(u);
}

macro observe_cc(u1, ..., uk, a1, ..., ak) {
bb_macro:
  opaque {
    v1 = mem[a1];
    ...
    vk = mem[ak];
    w1, ..., wk = snapshot(u1, ..., uk, v1, ..., vk);
    io(ordered_set_descriptor);
    yield();
  };
}
```

are not introduced via `tailio` patterns—which do not incur any ordering constraint—but define a relative ordering between different program properties (i.e. between different `observe_ccs`), as well as between a given property and different definitions of the observed values (i.e. different `artificial_defs`). These ordering constraints are cumbersome, and not always wanted by the programmer but rather a downside of the I/O-barrier-based approach.

5.2.3.5 Value Opacification

We have shown in Section 4.4.3 that it is possible to protect source-level security protections by preserving the protection-derived properties, which can be encoded as observational properties involving specific values key to the protections. As a result, a common pattern to protect security countermeasures consists in opacifying values, without modifying the data or control flow. In particular, rather than building an opaque chain of tokens—like `observe_decoupled` does—it is natural to chain observations using the original values but hiding them from potentially harmful transformations—i.e. building chains of opaque values produced by the `observe_opacify` pattern below.

```
macro observe_opacify(v1, ..., vk) {
bb_macro:
  u1 = opaque {
    w1, ..., wk = snapshot(v1, ..., vk);
    yield(w1);
  }
  return(u1);
}
```

This pattern implements the identity function on its first argument. Its other arguments can be used to express data dependence relations with other instructions. In addition, all arguments are observed. The compiler sees the result of `observe_opacify` as a statically unknown yet functionally deterministic value, it thus can assume that different observations (introduced via `observe_opacify` patterns) of the same input value v would yield the same output value u. In the following, any observation introduced via `observe_opacify` is called *opacification*.

Lemma 5.2.6 (Preservation of Value Opacification Events)**.** *If (1) all `snapshot` instructions of a program P are introduced via `observe_opacify`, if (2) the happens-before relation*

on opacifications in P is enforced through opaque chains whose tails are I/O events introduced via `observe_tailio`*, then opacifications in P are protected according to Definition 5.1.19 (Protected Observation).*

Notice that `observe_opacify` only differs from `observe_decoupled` in the returned value (the first argument rather than a token). As a consequence, the proof of Lemma 5.2.4 (Preservation of Decoupled Observation Events) applies directly to prove Lemma 5.2.6 (Preservation of Value Opacification Events).

Finally, to summarize different results shown in this section, the following theorem provides a methodology for protecting observation and opacification of any value(s) or address-value pair(s) throughout program transformations, i.e. preserving their existence as well as enforcing any partial ordering among these observations and opacifications.

Theorem 2 (Observation/Opacification protection)**.** *Let P be a program implementing observations and opacifications through a combination of* `observe_monolithic`*,* `observe_decoupled`*,* `observe_pair` *and* `observe_opacify` *in opaque chains enforcing a programmer-specified* $\xrightarrow{\text{hb}}$ *order, and such that any chain involving* `observe_decoupled`*,* `observe_pair` *and* `observe_opacify` *leads to a downstream transformation-preserved instruction (such as a trailing* `observe_tailio`*). Then all observations in P are protected according to Definition 5.1.19 (Protected Observation).*

Let P be a program implementing observations and opacifications through a combination of `observe_monolithic`*,* `observe_decoupled`*,* `observe_pair` *and* `observe_opacify` *in opaque chains enforcing a programmer-specified* $\xrightarrow{\text{hb}}$ *order, and such that any chain involving* `observe_decoupled`*,* `observe_pair` *and* `observe_opacify` *leads to a downstream instruction transformation-preserved conditionally on some instruction in a set* I_c*. Then all observations in P are protected conditionally on instructions in* I_c *according to Definition 5.1.19 (Protected Observation).*

The theorem above directly relates to observation and opacification preservation. However, more generally, the presented patterns can also be leveraged to support the evaluation of functional properties, which usually take the form of first-order logical formula. To this end, one may need to observe an arbitrary observation state with a variable number of $(name, value)$ pairs. The monolithic, decoupled and value opacification patterns above all achieve this, by collecting all variable names occurring in the logical property and atomically observing their name-value pairs. Nonetheless, the most adapted pattern for functional property preservation is the decoupled pattern, as it is lightweight (in contrast to monolithic pattern containing I/O instruction for every property) and "resource-less" (as it returns a token, which does not need hardware resources to actually store its value, in contrast to opacification pattern, which returns the opaque value requiring the allocation of a physical location).

Now that we have defined patterns implementing different observations and opacification schemes, specifying various constraints for compiler optimizations, the next section will describe how to implement some of these patterns in an optimizing compiler in practice. We deliberately chose to illustrate this idea on `observe_opacify` (for value observation and opacification), a variant of `observe_pair` (for memory observation), and `observe_tailio` (for opaque chain creation).

5.3 Putting it to Work

In this section, we first describe our proposed representation for observations in different program representation levels from source code, through compiler IR, down to binary code, then detail how we implemented some of the patterns, presented in Section 5.2.3, in the LLVM compilation infrastructure.

5.3.1 Observation and Opacification in Source Code

Recalling our first implementation, described in Section 4.3.1, which represents properties as annotations inserted in the source program. The compiler then transforms these annotations into observation point intrinsics (implementing the observations of observed variables and memory locations), and automatically inserts artificial definition intrinsics (implementing the opacifications of observed variables). Unlike this automated approach, we want to provide the programmers with a means to explicitly specify to the compiler the observations and opacifications of different values, directly in the source code.

To this end, we introduce language extensions to clang to support value observations and opacifications. We define three builtins `__builtin_opacify`, `__builtin_observe_mem` and `__builtin_io` corresponding, respectively, to the `observe_opacify`, a slightly different version of `observe_pair` and `observe_tailio` patterns defined in Section 5.2.3.

- `__builtin_opacify` is a variadic function implementing both value observation and opacification. It returns the same scalar value as its first argument, but made opaque to the compiler. This opaque value may then replace the original one in subsequent code (note that the replacement is no longer automatically performed during compile-time, but is specified by the programmers in the source program instead). All other arguments are optional and represent additional data dependence relations to implement opaque chains constraining program transformations. The builtin function also observes (snapshots) all its arguments: this provides a means to validate the opacification mechanism by tracing observed values down to the generated machine code.

- `__builtin_observe_mem` implements a memory observation. Unlike `observe_pair` pattern which observes a single memory location at a time, `__builtin_observe_mem` observes (reads) the whole memory region pointed to by its pointer-typed argument and returns a token to implement downstream opaque chains. This choice is driven by the "Sensitive Memory Data Erasure", described in Section 3.3.1), as the latter requires the observation of the whole secret buffer (cf. Section 5.4.1). Nonetheless, implementing `observe_pair` pattern would not particularly be a problem and is left for future work.

- `__builtin_io` is a variadic function implementing an I/O effect: the arguments serve to extend opaque chains linking upstream observations to the I/O effect. The function returns a token to implement downstream opaque chains.

These language extensions enable the programmer to define additional constraints when transforming the program, in the form of data dependences or ordering relations. As unit type is not natively defined in C, we currently use integer-typed variables that are only defined and used by our builtins to represent tokens produced by `__builtin_observe_mem` or `__builtin_io`. This has mainly two outcomes to the generated code. As will be detailed in Section 5.3.2.2, builtins (or more precisely, the

instructions corresponding to the high-level builtins) will be removed during code emission (recalling Section 4.3.4, where we also remove pseudo-instructions representing observation points and artificial definitions from the final machine code). If the removal of builtins eliminates all uses of a token variable, the latter is eliminated as well and does not incur any resource overhead. If the token variable remains live due to escaping values (in function call or return, or in memory), this variable, will incur low resource usage in the generated machine code, most likely a single stack slot for the whole function and no register usage beyond a short, temporary one during token definition on a *Reduced Instruction Set Computer* (RISC) *Instruction Set Architecture* (ISA). A better solution would be to implement a fully expressive token type in LLVM: the current one is limited—it cannot be used in ϕ nodes—and has a different, provenance-tracking purpose.

5.3.2 Observation and Opacification in LLVM

Let us now describe the transformation of our language extensions to two different compiler intermediate representations: the IR on which the optimizers operate and then the MIR which represents the final code to be emitted by the compiler.

5.3.2.1 Observation and Opacification in LLVM IR

As presented in Section 3.1.2, the LLVM IR supports intrinsic functions with compiler-specific semantics. Intrinsics require the compiler to follow additional rules while transforming the program.

To implement our preservation mechanism, we introduce three intrinsics to the LLVM IR:

- `llvm.opacify` has the same semantics as `__builtin_opacify`. More specifically, it opaquely produces a new SSA value from its first argument, so that compiler optimizations cannot reason about the relation between these two values, while other optional arguments produced by some preceding instructions ensure that the latter are scheduled before the intrinsic. All these argument values are also captured into the observation metadata (more on this later).

  Furthermore, the intrinsic is pure, does not access memory and has no I/O or other side-effects: it is valid to optimize away `llvm.opacify` if the opaque value is not used in subsequent code.

- `llvm.observe.mem` has the same semantics as `__builtin_observe_mem`. More specifically, it reads (an unspecified amount) from the memory that its pointer-typed argument points to. It returns a token represented as an SSA value.

  Unlike `llvm.opacify`, `llvm.observe.mem`'s attributes let it read argument-pointed memory. Other than such read, it has no I/O or other side-effects: it is valid to optimize away `llvm.observe.mem` if the output token is not used in subsequent code. Being able to access argument-pointed memory is actually an optimizing feature: this avoids having to generate instructions explicitly loading from these memory locations.

- `llvm.io` has the same semantics as `__builtin_io`. More specifically, it takes a variable number of SSA values as arguments and is defined as an I/O side-effecting instruction, so that it cannot be removed by optimizations, thus makes its argument values always live. It returns a token represented as an SSA value.

We modified clang to map `__builtin_opacify`, `__builtin_observe_mem` and `__builtin_io` to `llvm.opacify`, `llvm.observe.mem` and `llvm.io` respectively, when generating LLVM IR from C code.

Similarly to properties in our first implementation (cf. Section 4.3.4), observations are represented in LLVM IR as metadata. However, unlike property metadata, observation metadata is actually *attached* to the `llvm.opacify` or `llvm.observe.mem` intrinsic, instead of being passed as an operand of the intrinsic. Having no metadata operand allows the compiler to combine different observations of the same observed value. This is consistent with the semantics of opaque expressions defined in Section 5.2.1: the opaque values produced by these expressions are defined as statically unknown yet functionally deterministic values.

We also modify a few utility functions commonly used by different optimization passes such as `replaceAllUsesWith()` and `combineMetadata()` to update (e.g. when combining two intrinsics) and maintain (e.g. when duplicating the intrinsic) the metadata attached to the intrinsic throughout the optimization pipeline.

5.3.2.2 Observation and Opacification in LLVM MIR

To preserve values throughout code generation we also need to implement our mechanism in the MIR. We achieve this by lowering the intrinsics `llvm.opacify`, `llvm.observe.mem` and `llvm.io` respectively into `OPACIFY`, `OBSERVE_MEM` and `IO` pseudo-instructions, with the same semantics and behaviors w.r.t. memory accesses and side effects. Recalling pseudo-instructions are MIR instructions that do not have machine encoding information and must be expanded, at the latest, before code emission. Nevertheless, our mechanism should not interfere with the emitted machine code; the pseudo-instructions introduced are thus not expanded but completely removed during code emission. Similarly to artificial definition pseudo-instructions from Section 4.3.4, the `OPACIFY` pseudo-instructions uses the same register as its first operand to hold the opaque value, as a simple way to guarantee the correct functional behavior of the program when removing the pseudo-instructions,

The preservation of observation metadata is nonetheless more challenging: LLVM does not currently support attaching metadata to MIR instructions, we thus have to transform IR metadata into an operand of `OPACIFY` and `OBSERVE_MEM` pseudo-instructions. As discussed above, having metadata operands would prevent optimizations from combining pseudo-instructions with the same observed values but different observation metadata. As a result, this may require modifications to passes in the code generator to maintain and update the metadata while not preventing them from optimizing the program. Fortunately we did not have to do so since we did not find any such missed optimizations on our benchmark suite and on the different back-ends considered.

Note that, similarly to our I/O-barrier-based approach, the implementation of our I/O-barrier-free approach is fully compatible with LTO technique (cf. Section 4.3.4.2.

5.3.3 Observation and Opacification in Machine Code

At the final stage of the code generator, observation metadata is also emitted into the executable binary. In fact, this also allows to communicate information about the observed values to binary code utilities carrying out the validation of observation and opacification mechanisms (such as the debugger, binary code verifiers, etc.).

To represent this information in machine code, we extend the DWARF format, which provides an easily extensible description of the executable program (cf. Section 3.2). More specifically, we introduce to DWARF new tags and attributes to represent user-defined observations/opacifications. An observation from the source program is represented by a DIE which contains:

- the observation point, which is an attribute whose value is the address of the first executable instruction for the location identified by the builtin in the source program;
- references to the DIEs representing the observed values. Each of these DIEs contains the location attribute allowing binary code utilities to retrieve the values of these variables (cf. Section 3.2.2.1).

Similarly to the property DIE described in Section 4.3.2, observation/opacification DIE is owned by the subprogram DIE representing the function containing the observation point at which the observation/opacification occurs.

However, the biggest difference of this implementation compared to the more conventional approach, described in Section 4.3, which relies completely on debug information generated by the compiler itself is that we maintain and update the information of observed values ourselves. We track the operands of the observation/opacification intrinsics and pseudo-instructions throughout the whole compilation flow and modify the "code emission" phase of the compiler back-end so that not only observation/opacification DIEs are built from the corresponding metadata, but new dedicated DIEs, exclusively used to represent the observed values for a given observation/opacification, are also built and emitted during "code emission". The location attributes of these dedicated DIEs are computed using the tracked operands of the observation/opacification pseudo-instructions. In short, we only use DWARF for its standard encoding of the data, since it is already supported by most binary code utilities, and do not rely on the quality of debug information generated by the compiler.

5.4 Preserving Security Protections

It has been shown that there is a correctness-security gap in compilation, which arises when compiler optimizations preserves the functional semantics but violates a security guarantee made by source program. As a consequence, security engineers have been fighting with optimizing compilers for years by devising and introducing complex programming tricks to the source code, though yet found a reliable way to obtain secure binary code. In this section, we demonstrate how our lightweight observation/opacification preservation mechanisms can be used to preserve security protections through an optimizing compilation down to the generated binary. More specifically, we will illustrate our idea by considering use-cases covering the following security protection-derived properties, necessary to the effectiveness of the associated countermeasures:

- secret erasure property (i.e. proper erasure of sensitive data in memory, cf. Section 3.3.1);
- instruction ordering property (i.e. proper instruction ordering in masked secret key operations, cf. Section 3.3.2);

- code interleaving property (i.e. proper fine-grained interleaving of functional and protection code, cf. Section 3.3.3);
- redundancy preservation property (i.e. presence of redundant data and code to detect fault injections, cf. Section 3.3.4).
- constant-time selection property (i.e. presence of arithmetic idioms guaranteeing selection operation contains no jump or memory access pattern conditioned by the secret variable, cf. Section 3.3.5).

5.4.1 Sensitive Memory Data Erasure

We first start with the sensitive memory data erasure protection, described in Section 3.3.1, where a sensitive buffer on the stack must be zeroed with a call to `memset()` to avoid leaking confidential information in a cryptographic application. The code snippet shown in Listing 5.3a illustrates the scenario.

```
1
2
3
4
5  ...
6  void process_sensitive(void) {
7    uint8_t secret[32];
8    ...
9    memset(secret, 0, 32);
10
11 }
```

(a) Original attempt

```
1  #define OB_MEM(x)
2    __builtin_observe_mem(x)
3  #define IO(x)
4    __builtin_io(x)
5  ...
6  void process_sensitive(void) {
7    uint8_t secret[32];
8    ...
9    memset(secret, 0, 32);
10   IO(OB_MEM(secret));
11 }
```

(b) Using observation

Listing 5.3: Example of sensitive memory data erasure.

To preserve the erasure, we insert an observation (more specifically an opaque artificial read) of values stored in the buffer, after the call to `memset()`. We then use the value produced by the observation in an I/O effecting operation, as shown in Listing 5.3b. Therefore, a short opaque chain of length two links the observation `__builtin_observe_mem` to the final I/O builtin `__builtin_io`, guaranteeing that the former does not get removed (i.e the observation is protected, according to Definition 5.1.19 (Protected Observation)).

5.4.1.1 Mask Swapping Computation Order

Let us now consider the scenario of mask swapping. Masking of secret value seeks to ensure that all computations involving this value are statistically independent of it, thus avoiding leaking confidential information in a cryptographic application. To further maintain this statistical independence from the secret value, notably when only a limited set of masks is available, mask swapping is introduced to the program (cf. Section 3.3.2). The code snippet shown in Listing 5.4a illustrates the scenario.

The effectiveness of this mask swapping technique requires that the re-masking operation has to take place before the removal of the previous mask. Let us now illustrate how our mechanism can be used to force the compiler to respect this specific computation order.

In order to preserve the correct order in the mask swapping operation, we propose a solution based on opacification, as shown in Listing 5.4b. To prevent compiler optimizations from reodering the `^` operations, we opacify the result of the

```
1
2
3 ...
4
5 round_key[i] = (k[i] ^ mpt[i]) ^ m;
```

(a) Original attempt

```
1 #define OPC(x)
2   __builtin_opacify(x)
3 ...
4 round_key[i] = OPC(
5     OPC(k[i] ^ mpt[i]) ^ m);
```

(b) Using opacification

Listing 5.4: Respecting computation order in mask swapping operation.

re-masking operation, making it unknown to the compiler. The opaque value is next used in the removal of the old mask `m` (line 5). In order to form an opaque chain, the outer `OPC` (line 4) is introduced, making the definition of `round_key[i]` the tail opaque expression of the chain. There is no need for a terminal I/O builtin since we already know that the computation of `round_key[i]` is transformation-preserved, `round_key[i]` being the value of interest in downstream computation. Notice also that the cardinality constraint on values in opaque chains is trivially satisfied by the bijectivity of the `^` operator. The opaque chain enforces the ordering constraint that the opacified value of `k[i] ^ mpt[i]` will be computed before the unmasking operation.

As stated in Section 3.3.2, we validate our approach on a masked implementation of AES named `mask-aes`. In the following, we will also consider a self-written application called `mask-swap`, which contains a loop of mask swapping operations with the same security protection-derived property as `mask-aes`, together with I/O instructions; the goal is to evaluate the performance overhead of our lightweight mechanism relying on pure intrinsics without I/O side effects (i.e. I/O-barrier-free mechanism).

5.4.1.2 Step Counter Incrementation

We now consider the SCI countermeasure scheme that aims at detecting fault attacks that induce unexpected jumps to any location in the program (i.e. not executing at least two adjacent statements in of C source programs). The protection defines a step counter at each control construct, and steps the counter of the immediately enclosing control construct after every C statement of the original source. Counters are regularly checked against their expected values, calling a fault handler when it fails. An example of the SCI technique is illustrated in Listing 5.5a.

In order for the SCI countermeasure scheme to be effective, one needs to ensure the proper interleaving of original statements and counter incrementations and checks—i.e. ensure the code interleaving property—which, unfortunately, will be altered by compiler optimizations (cf. Section 3.3.3).

We make use of our opacification mechanism to preserve the SCI protection, as shown in Listing 5.5b. In fact, preserving the SCI protection boils down to (1) protecting counter incrementations and checks and (2) guaranteeing the proper interleaving of functional and countermeasure statements. The former can be achieved by opacifying counters at each of their incrementations (lines 11 and 15), so that counter values can no longer be deduced or constant-propagated and must be incremented instead. Note that in order to preserve the first check (line 9), we also have to opacify the initialization of `c_then` (line 6). Similarly, we opacify the initialization of `c_else` (line 7), so that the compiler always has to evaluate `c_else` when checking the boolean conditions involving the latter at the exit of the conditional statement (lines 18 and 20). As for guaranteeing the proper interleaving of functional and

```
1
2
3
4
5  ...
6  unsigned short c_then = 0;
7  unsigned short c_else = 0;
8  if (cond) {
9    if (c_then != 0)
10     killcard();
11   c_then++;
12   a = b + c;
13
14   if (c_then != 1)
15     killcard();
16   c_then++;
17 }
18 if (!((c_then == 2 &&
19         c_else == 0 && cond) ||
20        (c_then == 0 &&
21         c_else == 0 && !cond)))
22   killcard();
```

(a) Original countermeasure scheme

```
1  #define OPC(x)
2    __builtin_opacify(x)
3  #define OPC_ORD(x, y)
4    __builtin_opacify(x, y)
5  ...
6  unsigned short c_then = OPC(0);
7  unsigned short c_else = OPC(0);
8  if (cond) {
9    if (c_then != 0)
10     killcard();
11   c_then = OPC(c_then) + 1;
12   a = OPC_ORD(b, c_then) +
13       OPC_ORD(c, c_then);
14   if (OPC_ORD(c_then, a) != 1)
15     killcard();
16   c_then = OPC(c_then) + 1;
17 }
18 if (!((c_then == 2 &&
19         c_else == 0 && cond) ||
20        (c_then == 0 &&
21         c_else == 0 && !cond)))
22   killcard();
```

(b) Using opacification

Listing 5.5: Preserving code interleaving in SCI.

countermeasure statements, we create additional data dependences between values defined by the functional code (e.g. `a`) and counter values (e.g. `c_then`). To achieve this, we opacify non-constant operands used in definitions of functional values (e.g. `b` (line 12) and `c` (line 13) in the definition of `a`) and the condition expression used in the security check from the protection code (e.g. `c_then` from line 14) and express these artificial data dependences as token parameters of these opacifications, using a variant of `__builtin_opacify` named `OPC_ORD`. This creates an opaque chain linking every counter incrementation to the next counter use (in the functional code or in the security check), and then again to the next incrementation until the terminating fault handler `killcard()`—which is a transformation-preserved event—while interleaving original program statements in the chain through the bundling of both counter and original variables in opacification builtins. Notice that the opaque chain includes a control dependence when linking with the fault handler.

5.4.1.3 Control and Data Flow Redundancy

Next, we illustrate the preservation of the loop hardening scheme using our opacification mechanism. The loop protection consists in duplicating termination conditions and the computations involved in the evaluation of such conditions (cf. Section 3.3.4), in order to ensure that the hardened loop always performs the expected number of iterations and takes the right exit, even in the presence of fault attacks. Consider an implementation of the source-level loop protection on `memcmp()` function, shown in Listing 5.6a.

Listing 5.6b illustrates our proposed solution to preserve the redundancy preservation property during optimizing compilation. To prevent optimizations from removing the redundant data and code, we opacify every assignment to the duplicated variable by the use of `OPC` (lines 7 and 11): the compiler can no longer detect the identity relation between the original and its corresponding duplicated variable. Like in the previous example, the resulting opaque chains interleave original computations with checks, and link to a terminating fault handler through a control dependence.

```
1
2
3  ...
4  int memcmp(char *a1, char *a2,
5             unsigned n) {
6    unsigned i, i_dup;
7    unsigned n_dup = n;
8    for (i = 0, i_dup = 0;
9         i < n;
10        ++i, ++i_dup) {
11
12     if (i_dup >= n)
13       fault_handler();
14     if (a1[i] != a2[i]) {
15       if (a1[i_dup] == a2[i_dup])
16         fault_handler();
17       if (n_dup != n)
18         fault_handler();
19       return -1;
20     }
21   }
22   if (i_dup < n)
23     fault_handler();
24   if (n_dup != n)
25     fault_handler();
26   return 0;
27 }
```

(a) Original attempt

```
1  #define OPC(x)
2    __builtin_opacify(x)
3  ...
4  int memcmp(char *a1, char *a2,
5             unsigned n) {
6    unsigned i, i_dup;
7    unsigned n_dup = OPC(n);
8    for (i = 0, i_dup = 0;
9         i < n;
10        ++i, ++i_dup) {
11     i_dup = OPC(i_dup);
12     if (i_dup >= n)
13       fault_handler();
14     if (a1[i] != a2[i]) {
15       if (a1[i_dup] == a2[i_dup])
16         fault_handler();
17       if (n_dup != n)
18         fault_handler();
19       return -1;
20     }
21   }
22   if (i_dup < n)
23     fault_handler();
24   if (n_dup != n)
25     fault_handler();
26   return 0;
27 }
```

(b) Using opacification

Listing 5.6: Preserving redundancy in loop hardening.

5.4.1.4 Constant-Time Selection

Finally, we illustrate the preservation of the arithmetic idioms, widely-used in constant-time programming, that guarantees that the selection operation does not contain any jump or memory access pattern conditioned by the secret selection variable. However, it has been reported that these arithmetic idioms are not preserved by some LLVM back-ends, thus the code generated is not guaranteed to be constant-time (cf. Section 3.3.5). We consider different attempts at implementing constant-time selection from Listing 5.7a. The first two functions implement constant-time selection between two values `x` and `y` based on a secret selection boolean value `b`, while the third one illustrates a scenario where the programmer wishes to select a value from the lookup table `t` while hiding the secret lookup index `idx`. All these functions use a mask `m` to select the wanted value without conditional statement conditioned by the secret value (i.e. `b` or `idx`).

We propose in Listing 5.7b a solution to preserve the constant-time selection property relying on our opacification mechanism. The intuition is to hide from the compiler the correlation between the bitmask `m` and the secret values `b` and `idx`. This prevents the compiler from recognizing the selection idioms and turning it into conditional jumps: it embeds bitwise logic into an opaque chain linking selection arguments to the return value.

Moreover, we do not assume calls to these constant-time selection functions to be part of opaque chains; instead we create an opaque chain inside each function and make sure that it terminates by an opaque expression by opacifying the return value. This is an example of conditional transformation-preservation of instructions (cf. Definition 5.1.15 (Transformation-Preserved Instruction)): individual selection operations may or may not execute depending on (non-sensitive) program input, but as soon as one of these executes, the constant-time expressions it encloses will be

```
1
2
3  /// a. Constant-time selection
4  ///    between two values, v1
5  uint32_t ct_sel_val1(uint32_t x,
6                       uint32_t y,
7                       bool b) {
8    signed m = 0 - b;
9    return (x & m) | (y & ~m);
10 }
11
12 /// b. Constant-time selection
13 ///    between two values, v2
14 uint32_t ct_sel_val2(uint32_t x,
15                       uint32_t y,
16                       bool b) {
17   signed m = 1 - b;
18   return (x * b) |
19          (y * m);
20 }
21
22 /// c. Constant-time selection
23 ///    from lookup table
24 uint64_t ct_sel_lu(uint64_t t[8],
25                    size_t idx) {
26   uint64_t res = 0;
27   size_t i = 0;
28   for (; i < 8; ++i) {
29     bool c = (i == idx);
30     uint64_t m = (-(int64_t)c);
31     res |= t[i] & m;
32   }
33   return res;
34 }
```

(a) Original attempt

```
1  #define OPC(x)
2    __builtin_opacify(x)
3  /// a. Constant-time selection
4  ///    between two values, v1
5  uint32_t ct_sel_val1(uint32_t x,
6                       uint32_t y,
7                       bool b) {
8    signed m = OPC(0 - b);
9    return OPC((x & m) | (y & ~m));
10 }
11
12 /// b. Constant-time selection
13 ///    between two values, v2
14 uint32_t ct_sel_val2(uint32_t x,
15                       uint32_t y,
16                       bool b) {
17   signed m = OPC(1 - b);
18   return OPC((x * (1 - m)) |
19              (y * m));
20 }
21
22 /// c. Constant-time selection
23 ///    from lookup table
24 uint64_t ct_sel_lu(uint64_t t[8],
25                    size_t idx) {
26   uint64_t res = 0;
27   size_t i = 0;
28   for (; i < 8; ++i) {
29     bool c = (i == idx);
30     uint64_t m = OPC(-(int64_t)c);
31     res |= OPC(t[i]) & m;
32   }
33   return OPC(res);
34 }
```

(b) Using opacification

Listing 5.7: Preserving constant-time selection.

transformation-preserved due to the opaque chain forcing the compiler to compute the bitmask (as well as its complement for the first function) then using it for the selection.

Furthermore, for the third function, not only we want to ensure that the compiler does not transform the selection inside the loop into a branch conditioned by the secret index, but additionally we want to preserve the constant-timeness of the whole loop by making sure that the `|` operation takes place at every iteration. If the array contains a zero value, the compiler can assume that the result of the `&` operation will always be zero, regardless of the value of `m`, thus not performing the `|` operation to update `res`. As a result, we also opacify each element of the array.

It is worth noting that, unlike the traditional approach trying to reliably generate conditional move instruction whenever available, we accurately generate the expected constant-time code from the programmer's data-flow encoding of control flow. Although this may result in slower code, this can be directly applied to other constant-time operations involving value preservation.

Notice that for function `ct_sel_val2()` from Listing 5.7b, we also replace the multiplication `x * b` with `x * (1 - m)`. In fact, since `b` is of type `bool`, the compiler knows that `x * b` can only yield 0 or `x`, thus generates a conditional jump by enumerating all possible values of `b`. As `m` is opacified, the compiler does not know about the correlation between `m` and `b`, it is thus guaranteed that `1 - m` will be computed and used in the multiplication with `x`.

However, this example exposes the limit of our mechanism: we cannot rely on data opacification and dependences to prevent optimization passes from introducing control-flow to the program. To the best of our knowledge, there exists no real solution to this problem (yet): it has always been valid for compilers to modify the program's control-flow as long as this does not alter the program's behavior, and this is something we usually have no control over.

5.5 Validation

In this section, we describe the process used to validate our I/O-barrier-free approach and its implementation. We first validate the functional correctness of our approach and implementation, then describe our security validation methodology and use it to establish the preservation of the protections on the applications presented in Section 5.4.

5.5.1 Functional Validation by Checking Value Integrity and Ordering

According to Definition 5.1.18 (Observation-Preserving Transformation), establishing the preservation of an observation event amounts to proving the existence of an observation event (condition (ii)), for which all observed values are available (condition (iii)), at the proper memory address or associated with the appropriate variable, and following the specified partial order (condition (iv)). This consists in checking, for a given program execution, (1) the presence of all observation events, and (2) that for each of these events, the observed values of the specified variables and memory locations are the expected ones, and (3) that event ordering is compatible with the specified partial order. To this end, we leverage the concept of *observation trace*, which is the sequence of program observation states defined by all observation events encountered during a given execution of the program (cf. Definition 4.1.5 (Observation trace)). Similarly to the functional validation methodology used to validate our I/O-barrier-based approach, described in Section 4.4.1, the functional validation for our observation-preserving mechanism also involves comparing, for a given input of the program, two observation traces:

1. Reference trace: we execute the reference program compiled with optimizations disabled. The reference program is the original program (without our intrinsics) with `printf` inserted to generate the expected observed values. Again, we assume `-O0` preserves the observation events as well as the observation state of the ISO C abstract machine (ISO C11 Standard, 2011) containing the observed values of each event.

2. Optimized trace: we execute the program with builtins inserted, compiled with our compilation framework at different optimization levels. We modify a DWARF parser library (Eli Bendersky, 2019) to create a list of breakpoints containing all observation addresses in the binary code, as reported in the DWARF section. For each of these observation addresses, we record the locations where the observed values are stored. We finally retrieve these values during program execution, using a debugger.

To compare the traces, we associate each observation state defined by an observation event (`printf` in the reference program or intrinsic in our version) with a unique identifier—a combination of line and column numbers at which the event is defined

in the program source. We then verify using an offline validator (a small Python program) that (1) each observation state in the reference trace has a corresponding counterpart (having the same identifier) in the optimized trace, and inversely (2) each observation state in the optimized trace has a correspondence in the reference trace. The validator also verifies that all values in a given observation state from the optimized trace match the expected values reported in its reference counterpart. As a result, we refer to this validation scheme as *value integrity verification* in the remainder.

We apply the process described above to validate the functional correctness of our implementation on a subset of the test suite of Frama-C (Cuoq et al., 2012). This time, we restrict ourselves to properties verifying the expected values of variables at a given program point. These properties thus also include functional properties that we considered for the validation of our I/O-barrier-based approach (cf. Section 4.4.2.1) and can easily be expressed as observation events with our intrinsics.

Now, this leaves us with the question of validating observation ordering: unlike our I/O-barrier-based approach—which implicitly enforces a total ordering of program properties—we only enforce a partial ordering on observation events. More specifically, there is no particular constraint for the relative order of observation events having no dependence relation. Therefore, due to code motion or rescheduling during compiler optimizations, the reference and optimized traces might not be identical.

Furthermore, in the considered programs, there is no partial ordering constraint to be verified. As a consequence, we propose a validation methodology involving two different sets of programs. From the original program, for every function containing observation event(s), we derive (i) a totally ordered version where a unique temporary variable is used as a written-to and read-from token to chain all observation events (in other words, all events consume and write to the same token), and (ii) an unordered version where a distinct token is produced from every observation event and all these tokens are next consumed in a single, common I/O instruction at the end of the function (the I/O side-effects being decoupled from the observations themselves, it thus does not constrain the ordering of observation events that are not in dependence relation). The former is used to validate whether our implementation respects the simplest form of ordering, while the latter allows to verify the preservation of all observation events in presence of optimizations.

Moreover, since we model observation events as side-effect free, pure functions, different observations of the same values may be combined into a single one. As such, during program execution, a single point observing a given value might actually correspond to multiple observations of the same value. Although the transformation is perfectly valid, it leads to false positives reported by the validator when comparing observation traces. To eliminate these erroneous validation results, we update instruction combining support functions in LLVM to embed the metadata representing individual observations to be combined into a single combined observation (referencing both variable names, line numbers, etc.). These embedded observations will eventually be expanded when creating observation breakpoints for the debugger, which allows the corresponding observation states to be logged into the observation trace.

We compile each of these test cases at 6 optimization levels `-O0`, `-O1`, `-O2`, `-O3`, `-Os`, `-Oz`. This results in two sets of 31 applicable test cases featuring 616 observations—one set for totally ordered events (i) and one for unordered events (ii). Again, notice that these test cases are not meant to be evaluated as performance benchmarks, we only use them to validate the correctness of our implementation.

We automatically verified that in both sets, all 616 properties have been correctly propagated to machine code. In the first set (i) we checked that the observation trace is identical to the reference trace. In the second set (ii) we checked that values are all present and correct, even when observation events are transformed by compiler optimizations, but could not verify any partial ordering constraint (as there is none to be checked). All of this, at all considered optimization levels.

5.5.2 Security Protection Preservation Validation

Let us now survey the validation process of the preservation of different security protections described in Section 5.4. In fact, validating the preservation of security protections is more challenging. While verifying value integrity is enough to prove the preservation of observation events, it is only a necessary condition for preserving security protections. Hence, other than applying value integrity verification to the security applications, we also define additional mechanisms to validate specific components of the preserved protections.

Checking Value Utilization: In our security examples, opacification is used to (1) protect key values of the security countermeasure which are subject to program optimizations—such as duplicated variables from the loop hardening scheme or step counters from SCI protection—from being optimized away, and (2) make sure that the protected values are actually used in the subsequent code of the countermeasure (different security checks for instance). Clearly, value integrity verification only guarantees the former, we need a second verification to assess the latter. To this end, we perform a manual assembly code review in two phases. On the one hand, for each opacified value, we determine its uses in the binary program, then verify that these uses are indeed part of the original countermeasure in the source program. On the other hand, we identify in the binary program the critical parts of each of the security protections, such as the removal of the old mask from the masking operation, or various redundant checks. We then determine the operands of these key computations and verify that they indeed are results of opacifications.

This verification is undoubtedly protection-dedicated, as important uses of the opacified values really depend on the considered countermeasure scheme, and thus requires manual inspection of the generated code. Nonetheless, such verification of value utilization could be implemented as an automated two-phase data-flow analysis (with both backward and forward analysis) at the program's late MIR or machine code, as long as the analysis tool knows about uses of the opacified values crucial to the considered countermeasure.

Checking Statement Ordering: For SCI, other than preserving the step counter incrementations and the security checks, we also need to guarantee the proper interleaving of functional and countermeasure statements. This requires opacifying operands of every statement of the protected program with an artificial dependence of the result from previous statement, thus creating an opaque chain. We manually inspect the generated code to verify that, given two consecutive C statements S_1 and S_2, all MIR/assembly instructions between the opacification of the first evaluated operand of S_1 and the opacification of the first evaluated operand of S_2 correspond to S_1 (cf. Section 5.4.1.2).

Notice that there would be an option to automate this verification of the proper interleaving of functional and countermeasure statements, *if* one trusts the debug information to be sound and accurate. We could verify the line numbers, mapping

MIR/assembly instructions to the corresponding source statements and vice versa. If (1) all MIR/assembly instructions between the opacification of the first evaluated operand of a statement S_1 and the opacification of the first evaluated operand of the next statement S_2 have the same line number, and (2) during the scan over every MIR/assembly instructions of the program, the line number reported for each instruction (from the same basic block) is in an ascending order, it can be concluded that functional and countermeasure statements are correctly interlaced. Unfortunately debug information is not robust enough in general and we preferred to rely on manual inspection for higher confidence.

Checking Constant-Time Selection: As explained in Section 5.4.1.2, a widely-adopted, yet informal definition of constant-time selection is that there is no conditional branch based on a secret selection boolean value. To validate the preservation of constant-timeness using our opacification approach, we manually verify that none of the following three values is used to compute branch conditions: a secret selection boolean value, a bitmask created from it, or its opacified value. Notice that a conservative verification scheme could be implemented as a static data-flow analysis on the generated machine code, even though the automated determination of whether the branch conditions depend on the secret selection boolean value might be challenging, notably when the data flow involves memory accesses.

Validation Results: Table 5.1 summarizes the different validation schemes that we apply for each application presented in Section 5.4, in order to verify the preservation of different security countermeasures.

	`erasure-*`	`mask-*`	`loop-pin`	`sci-*`	`ct-*`
Value Integrity	✗✓	✗✓	✗✓	✗✓	✗✓
Value Utilization	N/A	✗✓	✗✓	✗✓	✗✓
Statement Ordering	N/A	N/A	N/A	✗✓	N/A
Constant-time Selection	N/A	N/A	N/A	N/A	✗✓

TABLE 5.1: Validation of different security applications. ✗ indicates the scheme is *applied* to the program, N/A indicates the scheme is not relevant to the program. ✓ indicates the scheme is *validated* for the program.

For each application we verified that the appropriate schemes yield the expected results, validating our approach and implementation.

It is worth noting that we cannot directly leverage binary security analysis to validate the preservation of security protections throughout the optimizing compilation flow. In fact, the goal of security analysis is to evaluate the effectiveness of a given security protection against some threat models, while our preservation mechanism only seeks to guarantee that the generated binary implements the same security protection introduced in the source program and does not pretend to transform an ineffective security protection from the source program into a more secure one in the binary. In fact, properly assessing the effectiveness of security protections, such as comparing the program's robustness against fault injections between an unoptimized version (compiled at `-O0`) and an optimized with protection-preserved version (compiled with our compiler at any optimization level) would be interesting and is actually left for future work. As a result, in order to validate the correctness of our I/O-barrier-free approach and its implementation in LLVM, we deliberately

devise the different validation schemes described in this section and verify the presence of the security protections in the generated binary programs.

5.6 Experimental Evaluation

Let us now evaluate our I/O-barrier-free mechanisms on the security applications presented in Section 5.4, where we analyze the impact of our mechanisms on the program's performance and compilation time.

5.6.1 Experimental Setup

For each considered security applications, we first compare our opacified versions against the unoptimized programs—which is also a solution to preserving security protections—to quantify performance benefits.

We then compare our opacified versions against other available preservation mechanisms, namely compiler-dependent programming tricks—that are currently implemented in popular cryptographic libraries such as OpenSSL or mbedTLS—for constant-time selection (Simon, Chisnall, and Anderson, 2018), and our I/O-barrier-based approach for all other applications (cf. Chapter 4).

Eventually, we also compare our implementation of the I/O-barrier-free mechanism described in Section 5.3 with an alternative one that does not involve modifications to clang and LLVM but relying on inline assembly instead. This will be described in details in Section 5.6.4. In the following, we will refer to the former as *compiler-native implementation*, and the latter as *inline-assembly-based implementation*.

Finally, we present the compilation time overhead of our implementation.

For all benchmarks, we target two different instruction sets: ARMv7-M/Thumb-2 which is representative of deeply embedded devices, and Intel x86-64 representative of high-end processors with a complex micro-architecture. In addition, since it has been reported that the compilation of source-level constant-time selection code on the IA-32 architecture contained secret-dependent conditional jumps (Simon, Chisnall, and Anderson, 2018), we also consider IA-32 for the constant-time applications `ct-rsa` and `ct-montgomery`.

Performance evaluation for the ARMv7-M/Thumb-2 ISA takes place on an MPS-2+ board with a 32-bit Cortex-M3 clocked at 25 MHz with 8 Mb of SRAM, while our Intel test bench has a quad-core 2.5 Ghz Intel Core i5-7200U CPU with 16 GB of RAM. Note that for the ARMv7-M/Thumb-2 ISA, we also measure the program performance in terms of number of instructions executed, using ARM Fast Models (cf. Section 4.2), and obtain basically the same result (i.e. same performance ratio) as comparing program execution time. This thus confirms the relevance of measuring performance in terms of number of instructions executed from Section 4.2, given the Cortex-M3's very simple pipeline.

We use the Intel platform for compiling for either target. Changing the target only concerns the back-end, a short part of the compilation pipeline, as a result, we only report the compilation time evaluation results for the ARMv7-M/Thumb-2 ISA.

Our experiments cover all common optimization levels (`-O1`, `-O2`, `-O3`, `-Os`, `-Oz`). Performance measurements are based on the average of 10 runs of each benchmark and configuration.

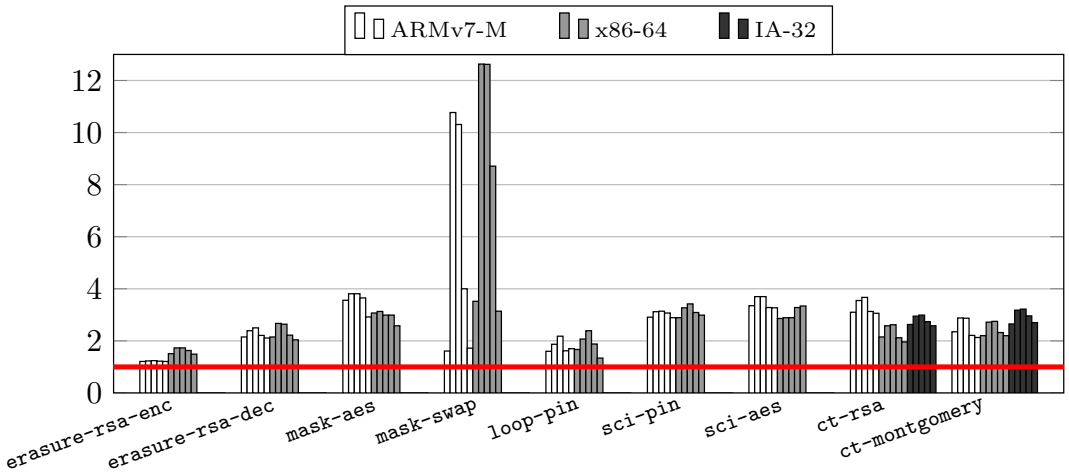

FIGURE 5.3: Speed-up of our approach over unoptimized original programs—ordered by compiler option `-O1`, `-O2`, `-O3`, `-Os`, `-Oz`. The horizontal red line represents a performance ratio of 1.

5.6.2 Comparing to Unoptimized Programs

Figure 5.3 presents the speed-up of our approach at different optimization levels over unoptimized programs. For all benchmarks, speedup ranges from 1.2 to 12.6, with an harmonic mean of 2.8. Clearly, our observation- and opacity-based approach to preserving security protections enables aggressive optimizations with significant benefits over `-O0`.

5.6.3 Comparing to Reference Preservation Mechanisms

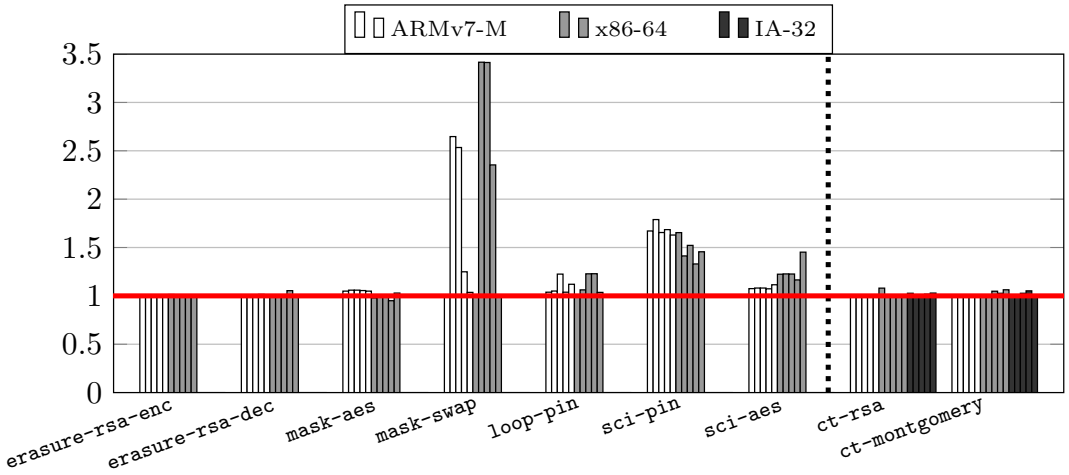

FIGURE 5.4: Speed-up of our approach over reference preservation approaches—I/O-barrier-based property-preserving mechanism (Chapter 4) for applications on the left side of the dotted line and programming tricks (Simon, Chisnall, and Anderson, 2018) for the ones on its right side—ordered by compiler option `-O1`, `-O2`, `-O3`, `-Os`, `-Oz`. The horizontal red line represents a performance ratio of 1.

Figure 5.4 presents the speedup our approach compared to reference preservation approaches, at different optimization levels. For `erasure-rsa-enc`, `erasure-rsa-dec`, `mask-aes`, `mask-swap`, `loop-pin`, `sci-pin` and `sci-aes`, we compare our approach against our I/O-barrier-based property preservation mechanism described

in Chapter 4. Recalling we also introduced new intrinsics to implement their property preservation mechanism, however, we relied heavily on the I/O side effects of the intrinsics: not only we introduced I/O side-effecting intrinsics to model observation points so that these cannot be removed by optimizations, we also inserted I/O side-effecting artificial definitions for every property-observed value to protect these from being optimized out. Moreover, to ensure the correct values in memory at observation points, the former intrinsics also behave like memory fences, i.e. can read from and write to memory. Furthermore, in order to guarantee the correct debug information for these values, we inserted even more artificial definitions to prevent multiple live ranges corresponding to the same source variable from overlapping.

As a consequence, our I/O-barrier-free approach with pure intrinsics (no side-effects), accessing memory only when required, should enable more optimizations and thus result in faster code. Our results clearly confirm this. For example, although `mask-swap` contains the masking computation, the data used in the operation is passed as function arguments instead of being declared as global variables in reference implementations; this clearly allows more optimizations when the function is inlined (i.e. when compiled at `-O2`, `-O3` or `-Os`), and especially when the function call is inside a loop. More generally, optimizations such as "loop unrolling" and "loop invariant code motion" are the main sources of benefits at these optimization levels. On the contrary, for `mask-aes`, the data required for the masking computation is stored in memory as global variables, there is almost no difference between two versions: the masking operation contains loads and stores to the secret key as well as the different masks in the order defined in the source program, as the protection is correctly preserved.

As for `erasure-rsa-enc` and `erasure-rsa-dec`, the function implementing the protection only contains the erasure of the sensitive buffer, we thus observe almost no difference compared to our I/O-barrier-based approach.

As for other applications (`loop-pin`, `sci-pin` and `sci-aes`), for both targets, we note a clear improvement, ranging from 1.04 to 1.79, with an harmonic mean of 1.3. Overall, I/O side-effecting intrinsics restrict compiler optimizations, thus inevitably degrade the performance of generated code.

For `ct-rsa`, we compare our approach against the constant-time selection implementation of mbedTLS (Bakker, 2019), which is basically the same as the version from Listing 3.9a.a, but with the computation of the bitmask (line 3) splitted into a separate function. Furthermore, this function must not be inlined in order to prevent the compiler from optimizing it away. As for `ct-montgomery`, we compare our approach against the specially-crafted implementation of OpenSSL (The OpenSSL Project, 2003). It is worth noting that general-purpose compilers offer no guarantees of preserving constant-timeness: future versions of the same compiler may spot the trick and optimize the constant-timeness away (Simon, Chisnall, and Anderson, 2018). Although our approach allows the functions implementing constant-time selection to be inlined while still preserving constant-timeness, these only take a small fraction of the execution time; we do not notice a clear difference compared to other constant-time implementations.

5.6.4 Comparing to Alternative Implementations

Given our I/O-barrier-free observation-preserving mechanism, let us now survey the alternative inline-assembly-based implementation to our compiler-native implementation described in Section 5.3.

Compilers provide an *inline assembly* syntax to embed target-specific assembly code in a function. The feature is regularly used by C programmers for low-level optimizations and operating system primitives, and also for sensitive applications to avoid interference from the compiler (Rigger et al., 2018). gcc-compatible compilers implement an extension of the optional ISO C standard syntax for inline assembly, allowing programmers to specify inputs or outputs for inline assembly as well as its behavior with respect to memory accesses and I/O effects (Stallman and Community, 2009). Compilers only care about this specification when performing code transformation on the inline assembly and are completely agnostic to what happens *inside* the inline assembly region. In other words, the assembly code region is opaque to the compiler. We may thus leverage this feature to implement opaque expressions. For example, to preserve the correct masking order in Listing 5.4b, the call to `__builtin_opacify` at line 4 may be replaced by an inline assembly expression, as shown in Listing 5.8.

```
1 k ^= m;
2 ...
3 uint8_t tmp = k ^ mpt;
4 __asm__ ("" : "+r" (tmp));
5 k = tmp ^ m;
```

LISTING 5.8: Secure masking using opacification based on inline assembly.

The inline assembly region (line 4) is actually empty: the behavior exposed to the compiler of the whole expression is specified by the `"+r" (tmp)` constraint. This essentially means that `tmp` is both the input and output of the expression, and that the expression neither accesses memory nor does it have any side-effect. As an output of the inline assembly expression, `tmp` is now opaque to the compiler, just as if it was defined by the `__builtin_opacify`.

This example can be generalized to implement any opaque region. In practice, it is sufficient to implement a small set of builtins covering the typical opacification scenarios. A set of preprocessor macros can be designed to cover these typical scenarios and thus can be provided as a portable interface across most compilers and targets.

Note that this approach slightly complicates the implementation of observations, carrying precise variable names, memory addresses, line numbers down to machine code. Additional conventions and post-pass on the generated assembly code are required to produce the appropriate DWARF representation, as described in Section 5.3.3.

Now, the natural question is to compare the performance of an inline-assembly-based implementation with our compiler-native opaque regions. To this end, we consider a subset of the applications presented in Section 3.3, containing `erasure-rsa-enc`, `erasure-rsa-dec`, `mask-aes`, `mask-swap`, `loop-pin`, `ct-rsa` and `ct-montgomery`. We exclude `sci-pin` and `sci-aes`, as these applications would require the manual insertion of inline assembly expressions at every statement of C source programs, which is impractical. Figure 5.5 presents the speedup of our compiler-native implementation w.r.t. inline-assembly-based implementation, at different optimization levels.

In the majority of cases, the two considered implementations generate the same executable code. For `ct-rsa` and `ct-montgomery`, there is a slight difference in performance due to discrepancies in register allocation. This is not visible in other

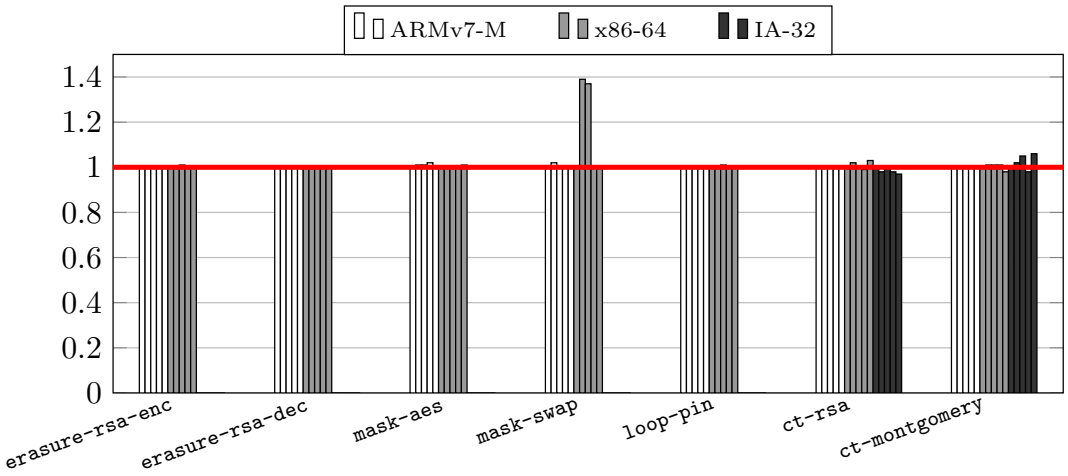

FIGURE 5.5: Speed-up of our compiler-native implementation over inline-assembly-based implementation—ordered by compiler option `-O1`, `-O2`, `-O3`, `-Os`, `-Oz`. The horizontal red line represents a performance ratio of 1.

applications because these two programs are larger and demonstrate higher register pressure. The only significant performance difference is for `mask-swap` compiled with `-O2` and `-O3` for x86-64: our compiler-native implementation is 40% faster than inline assembly. The core loop of the inline-assembly-based version happens not to be unrolled, while compiler-native version's is. Interestingly, this is only the case for x86-64: the same loop is unrolled for both versions when compiling for ARMv7-M. Indeed, the difference disappears when we force loop unrolling using the `-funroll-loops` option together with `#pragma unroll`. As expected, inline assembly occasionally interferes with compiler optimizations, despite the precise specification enabled in its syntax, while compiler intrinsics allow for carrying more precise semantics to the optimizers. Mitigations exist, and make the inline assembly approach interesting to some multi-compiler development environments. The take away from this is that both approaches are sound and leverage the same formalization and secure development scenarios (for opacification purposes, not for observation purposes). Yet this may not always be the case in the future: compilers are not forbidden to analyze inline assembly and take optimization decisions violating the opacity hypothesis; the fact they do not do it today is no guarantee that secure code will remain secure in future versions. On the contrary, our intrinsics in the source language and IR have a explicit and future-proof opacification and observation semantics.

5.6.5 Compilation Time Overhead

Figure 5.6 shows the compilation time overhead compared to compiling the original programs at the same optimization level. Note that the optimized original programs are insecure, as protections have been stripped out or altered by optimizations. We consider the Intel platform since it is used for both native and cross-compilation for both targets.

In general, the compilation time overhead is insignificant (under 7%). Note that in certain cases, the compilation time is even smaller, although marginally, when compared to compiling the original programs at the same optimization level. This is due to the imprecision in our time measurements on Intel platform. As for `sci-aes` and `sci-pin`, the overhead ranges from 13% up to 70%. As discussed in Section

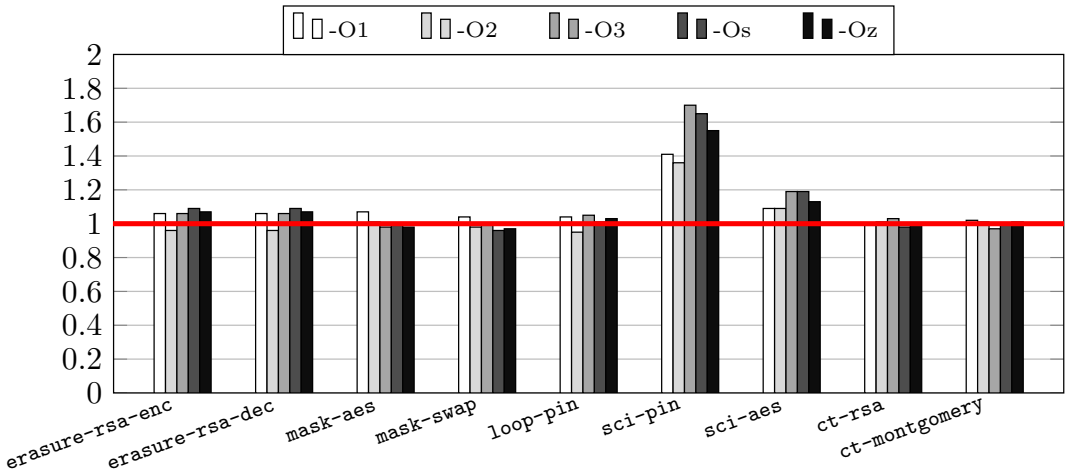

FIGURE 5.6: Compilation time overhead on the Intel platform, compared to compiling the original programs at the same optimization level. The horizontal red line represents a performance ratio of 1.

4.4.4.2, the SCI protection represents a very important part of the whole application (as counter incrementations are inserted after each C instruction), and it is completely stripped out from original programs when optimizations are enabled without opacification. As a consequence, the code size of the insecure baseline is much smaller than the secure code with fully-preserved countermeasures, which justifies the important compilation time difference.

5.7 Discussion

In this chapter, we formally defined the notion of observation and its preservation through program transformations. We instantiated this definition and preservation mechanisms through multiple program representations, from C source code down to machine code. The approach relies on two fundamental principles of compiler correctness: (1) the preservation of I/O effects and (2) the interaction of data dependences with program constructs that are opaque to static analyses. We also formally proved the correctness of the approach on a simplified intermediate language, and implemented it within the latest LLVM compilation infrastructure, with only minimal change to existing compilation passes (more precisely, to some common utility functions used by these passes).

We apply our approach to specifically address a fundamental open issue in security engineering, namely the preservation of source-level security protections throughout the optimizing compilation flow. We validated our approach and its implementation on a set of use-cases involving different security protections, notably against side-channel and fault injection attacks. Nonetheless, perhaps the biggest shortcoming of our work described in this chapter is the lack of automation in the security validation process, which relies heavily on manual inspection of binary code. Unfortunately, as discussed in Section 5.5.2, the security validation process is really sensitive to the considered protection schemes; it is thus complicated to design a generic and automated validation process for all protections. This is actually an important problem that should be further studied, and will be discussed in the next chapter. Another problem that we need to deal with is the relatively modest number of examined security protections. This does not mean that our approach is unable

to preserve other protections, but rather because collecting security protections suffered from compiler optimizations is tedious and time-consuming, and we did not have enough manpower to work on this issue.

Despite our emphasis on security in the considered examples and experimental evaluation, the problem we consider as well as our proposed solution may have general applications in software engineering. We will discuss these avenues for further research in the next chapter.

Chapter 6

Conclusion

6.1 Conclusion

Target programs produced by an optimizing compilation process may present vulnerabilities, even when the source program was assumed or proven secure with regards to a given security property. Therefore, either the security properties need to be preserved along the compilation flow, or they need to be verified in compiled programs—which requires the compiler to convey additional information (e.g. functional properties describing the program's expected behaviors) to these security verifications and analyses—or both. Typically, security properties are enforced by introducing countermeasure schemes at the source-level to protect the programs against specific threat models. Interestingly, these protections can be preserved if compilers manage to maintain the protection-derived properties associated with such protections.

As a result, modern compilers nowadays should not only perform mere translation from a source language to a target language, but also have to be responsible for carrying security-related properties (i.e. functional properties to carry out security analyses at binary level or protection-derived properties required for security protection preservation) to the target program, and this is becoming even more critical over time.

Unfortunately, compiler optimizations—which focus solely on achieving better program performance, typically by reorganizing computation and removing "unnecessary" code—are not suitable by design for preserving the security-related properties. The difficulty primarily lies in the fact that this information usually cannot be explicitly specified in the high-level source language, thus compilers have no notion of the link between the extra properties and the code they refer to; they have no means to constrain transformations to preserve this link or to update the properties to adjust to any code transformation.

At root, this thesis is devoted to tackling this challenge by first investigating the interaction between code optimization and security-related property preservation, then by proposing compilation mechanisms to maintain and propagate security-related properties down to machine code. Furthermore, one of our goals consists in integrating our mechanisms into a widely-used production compiler, to contribute to the security and compilation communities.

To this end, we first define the notion of functional property and its preservation in compilation, and propose an I/O-barrier-based approach to preserve these properties across all program representations through the optimizing compilation of C to machine code. This approach relies on the insertion, into the intermediate programs throughout the compilation process, of opaque and I/O side-effecting instructions that the compiler cannot analyze, and thus they cannot be removed by optimizations. These instructions are removed during code emission and not emitted to the

final executable binary, thus minimizing the interference with the original program. We further leverage this functional property preservation mechanism to preserve security protection-derived properties—which, in general, can be equivalently encoded as observational properties, a special type of functional properties—thus preserving the associated security protections throughout optimizing compilation flow.

Although we tried to minimize the negative impact of property preservation on compiler optimizations, our I/O-barrier-based approach unintentionally, by design, induces a total ordering, during program transformations, of these instructions, relative to each other and relative to other I/O side-effecting instructions of the original program. However, this ordering constraint is not required by the preservation itself, and is not even always desirable, as it might hamper the generated code's performance. As a result, we naturally seek to answer the question of whether it is possible to lift the ordering constraint with a more fine-grained approach.

For that purpose, we first start with the remark that the problem of preserving functional properties (and thus observational properties) can be reduced to the preservation of the observations of specific program values at program points corresponding to where the properties occur in the source program. We then formally define the notion of observation and its preservation through program transformations and illustrate these notions on Mini IR, a simplified intermediate program representation. Next, we propose an I/O-barrier-free mechanism to preserve observations down to machine code, which solely leverages the most essential information modeled by nearly every compiler transformation: I/O side-effects and data dependences. We control this information according to the specified observations through the ability to hide information about an atom of operational semantics—we call this technique opacification. Moreover, we formally prove the correctness of our mechanism on Mini IR.

We implement both approaches in the LLVM framework, showing that preserving properties and observations does not necessarily require significant modifications to compilers. Furthermore, we extend the DWARF debug information standard to encode the preserved properties or observations in the executable binary, so that binary analysis tools or validators can retrieve these and carry out their verifications and analyses.

We validate our approaches and implementations on a range of security use-cases, featuring various source-level protection schemes, expressing the protection-derived properties in terms of observations to be made at specific points of the computation. The result shows that our approaches provide a reliable means to preserve security-related properties throughout the compilation process, while still leaving freedom to the compiler to perform aggressive optimizations. As such, our proposals address the fundamental open issue in security engineering of preserving source-level security countermeasures through optimizing compilation flow.

6.2 Perspectives

The primary focus of this thesis is to demonstrate the effectiveness of our property preservation mechanisms for preserving source-level security protections. However, property preservation can be used for far more than just security-related applications and we have not exhausted the scope of our mechanisms even within this area. In this section, we want to describe avenues for further research in the continuation of the work presented in this thesis.

For security-related applications of our work, the most legitimate area of improvement consists in designing automated security validation process in order to approve, with ease and high confidence, the preservation of security protections by our compilation framework. We provided some suggestions on this topic in Section 5.5.2, by hinting at devising data-flow analysis, at late-MIR or binary level, to verify that opacified values are actually used in subsequent code, or there is no conditional branch based on a secret selection boolean value for example. Notice that in order to carry out such analysis, the compiler (if the analysis is implemented at late-MIR level) or the binary analysis tool (if the analysis is implemented at binary level) requires additional information such as the opacified values, the important uses of such values, or the secret selection boolean value. Interestingly, this extra information can easily be determined and expressed at the source level (as programmers are most comfortable working with denotations defined in source programs), and our property-preserving compilation framework—which is designed to maintain and propagate extra information—can be leveraged to convey the extra information to the appropriate tool.

Another intriguing research direction in security-related applications of our work consists in formalizing the idea of preserving source-level security protections by identifying the associated protection-derived properties and encoding the latter as observations, which in turn are guaranteed to be preserved by our proposed (i.e. I/O-barrier-based and I/O-barrier-free) mechanisms. More specifically, it would be interesting to answer the question of whether *every* security protection can be preserved using this approach and if not, what would be the criteria for a protection to have an associated protection-derived property that can be encoded as observation. This will allow for a classification of security protections and more importantly, provide a systematic approach to encode and translate security protections into equivalent ones preservable using our proposed preservation mechanisms, whenever possible. As a consequence, this naturally eases the process of collecting more use-cases where security protections can be preserved by our mechanisms.

Other than preserving source-level protections, our mechanisms can easily be leveraged to protect security countermeasures introduced at compile-time, such as the data integrity protection (cf. Section 2.2.4.1) or the original loop protection (cf. Section 2.2.4.3), from being altered by downstream transformations in the compilation pipeline. Furthermore, our mechanisms also enable the implementation of novel countermeasure schemes. For instance, to completely mitigate the problem of secret erasure caused by "dead store elimination", not only the secret buffer has to be erased before the return of the sensitive function, all potential copies of the secret data also have to be cleared. Recent work has proposed to systematically erasing the whole stack frame and registers used in the sensitive function (Simon, Chisnall, and Anderson, 2018). However, as pointed out in Section 2.2.3.3, this may not be the most efficient approach and can be improved by implementing a post-register-allocation transformation that selectively erases stack slots and registers containing a copy of the secret data at the end of the function. To this end, extra information about the secret data is required, and this is exactly where our property preservation mechanisms step in to provide the transformation with this information.

Concerning non-security-related applications of our work, an interesting trajectory is to improve debugging and unit testing. In fact, source-level properties can be a powerful debugging tool, helping enforce or detect violations through the development and deployment process. This is similar to inserting (runtime) assertions, except that the code does not have to carry these runtime checks when running in production. Instead, the ability to propagate source-level properties down to machine

code, allows a debugger to trace program execution and evaluate the properties in a testing environment. This brings the promise of using the same executable code in both testing and production environments. Unlike assertions, it is expected that machine code annotations do not consume execution time. On the contrary, their preservation through the compilation flow is not easy (as we have seen earlier in secure applications) and may also force the compiler to occasionally restrain its full optimization potential. We are not aware of any software engineering approach being currently pursued together with an aggressively optimizing compiler, precisely because optimizations are prone to destroying the link between the high-level properties and the machine code where they are meant to be tested. And indeed, while these are common issues related to the propagation of debug information through optimization passes, the usual tradeoff in such a case is to preserve debug information only as a best-effort strategy, which is insufficient for our debug-only-assert scenario. Our observation and opacification techniques allow to handle exactly this issue and thus can be leveraged to improve the debugging and testing process.

Personal References

The work described in this thesis has been subjected to the following publications and presentations:

Preliminary Work

- Son Tuan Vu, Karine Heydemann, Arnaud de Grandmaison, Albert Cohen. “Compilation et optimisation de code en présence d’annotations de sécurité”, *13ème Journée de Compilation*, Jan. 2019, Dammarie Les Lys, France.
- Son Tuan Vu, Karine Heydemann, Arnaud de Grandmaison, Albert Cohen. “Compilation and Optimization with Security Annotations”, *European LLVM Developers Meeting*, Apr. 2019, Bruxelles, Belgium.

Chapter 4

- Son Tuan Vu, Karine Heydemann, Arnaud de Grandmaison, Albert Cohen. "Secure delivery of program properties through optimizing compilation". *CC ’20: 29th International Conference on Compiler Construction*, Feb. 2020, San Diego, CA, United States. pp.14-26, doi: 10.1145/3377555.3377897.
- Son Tuan Vu, Karine Heydemann, Arnaud de Grandmaison, Albert Cohen. “Secure Delivery of Program Properties with LLVM ”, *European LLVM Developers Meeting*, Apr. 2020 (cancelled), Paris, France.
- Son Tuan Vu, Karine Heydemann, Arnaud de Grandmaison, Albert Cohen. “Propagation et préservation de propriétés dans un flot de compilation optimisant et applications à la préservation de protections contre les attaques en faute”, *Journée thématique sur les Attaques par Injection de Fautes*, Sept. 2020, Paris, France.

Chapter 5

- Albert Cohen, Son Tuan Vu, Karine Heydemann, Arnaud de Grandmaison, Christophe Guillon. “Secure Optimization Through Opaque Observations”, *Workshop on Principles of Secure Compilation*, Jan. 2021, Online.
- Son Tuan Vu, Albert Cohen, Karine Heydemann, Arnaud de Grandmaison, Christophe Guillon. “Secure Optimization Through Opaque Observations”, Jan. 2021. arXiv:2101.06039 [cs.CR]
- Son Tuan Vu, Albert Cohen, Karine Heydemann, Arnaud de Grandmaison, Christophe Guillon. “Secure Optimization Through Opaque Observations”, *Journée Groupe de Travail Méthodes Formelles pour la Sécurité*, Mar. 2021, Online.

Furthermore, the work in Chapter 5 is also being submitted for a journal article, under the title “Reconciling Observation With Optimization in Secure Compilation”.

Bibliography

[AB08] Amal Ahmed and Matthias Blume. "Typed Closure Conversion Preserves Observational Equivalence". In: *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*. ICFP '08. Victoria, BC, Canada: Association for Computing Machinery, 2008, 157–168. ISBN: 9781595939197. DOI: 10.1145/1411204.1411227. URL: https://doi.org/10.1145/1411204.1411227.

[Aba+19] Carmine Abate, Roberto Blanco, Deepak Garg, Catalin Hritcu, Marco Patrignani, and Jérémy Thibault. *Journey Beyond Full Abstraction: Exploring Robust Property Preservation for Secure Compilation*. 2019. arXiv: 1807.04603 [cs.PL].

[Aba99] Martín Abadi. "Protection in Programming-Language Translations". In: *Secure Internet Programming: Security Issues for Mobile and Distributed Objects*. Ed. by Jan Vitek and Christian D. Jensen. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 19–34. ISBN: 978-3-540-48749-4. DOI: 10.1007/3-540-48749-2_2. URL: https://doi.org/10.1007/3-540-48749-2_2.

[Abs] AbsInt. *aiT*. https://www.absint.com/ait/index.htm. Accessed 19 May 2018.

[AFG00] Martín Abadi, Cédric Fournet, and Georges Gonthier. "Authentication Primitives and Their Compilation". In: *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '00. Boston, MA, USA: Association for Computing Machinery, 2000, 302–315. ISBN: 1581131259. DOI: 10.1145/325694.325734. URL: https://doi.org/10.1145/325694.325734.

[Aho+06] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2Nd Edition)*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2006. ISBN: 0321486811.

[AP12] Martín Abadi and Gordon D. Plotkin. "On Protection by Layout Randomization". In: *ACM Trans. Inf. Syst. Secur.* 15.2 (July 2012). ISSN: 1094-9224. DOI: 10.1145/2240276.2240279. URL: https://doi.org/10.1145/2240276.2240279.

[App98] Andrew W. Appel. "SSA is Functional Programming". In: *ACM SIGPLAN Notices* 33.4 (1998), pp. 17–20. DOI: 10.1145/278283.278285. URL: https://doi.org/10.1145/278283.278285.

[ARM19] ARM. *ARM Fast Models*. 2019. URL: https://developer.arm.com/tools-and-software/simulation-models/fast-models (visited on 08/14/2019).

[AS85] Bowen Alpern and Fred Schneider. "Defining Liveness". In: *Inf. Process. Lett.* 21 (1985), pp. 181–185.

[Bak19] Paul Bakker. *mbedTLS*. Version 2.17.0. Mar. 19, 2019. URL: tls.mbed.org.

[Bal+10] Clément Ballabriga, Hugues Cassé, Christine Rochange, and Pascal Sainrat. "OTAWA: An Open Toolbox for Adaptive WCET Analysis". In: *Software Technologies for Embedded and Ubiquitous Systems*. Ed. by Sang Lyul Min, Robert Pettit, Peter Puschner, and Theo Ungerer. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 35–46. ISBN: 978-3-642-16256-5.

[Bar+19] 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 (Dec. 2019). DOI: 10.1145/3371075. URL: https://doi.org/10.1145/3371075.

[Bau+08] Patrick Baudin, Jean C. Filliâtre, Thierry Hubert, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. *ACSL: ANSI/ISO C Specification Language Version 1.4*. May 2008. URL: https://frama-c.com/download/acsl.pdf.

[Bay+13] Ali Galip Bayrak, Francesco Regazzoni, David Novo, and Paolo Ienne. "Sleuth: Automated Verification of Software Power Analysis Countermeasures". In: *Cryptographic Hardware and Embedded Systems - CHES 2013*. Ed. by Guido Bertoni and Jean-Sébastien Coron. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 293–310. ISBN: 978-3-642-40349-1.

[BCR16] Thierno Barry, Damien Couroussé, and Bruno Robisson. "Compilation of a Countermeasure Against Instruction-Skip Fault Attacks". In: *Proceedings of the Third Workshop on Cryptography and Security in Computing Systems*. CS2 '16. Prague, Czech Republic: ACM, 2016, pp. 1–6. ISBN: 978-1-4503-4065-6. DOI: 10.1145/2858930.2858931. URL: http://doi.acm.org/10.1145/2858930.2858931.

[BDJ18] Frédéric Besson, Alexandre Dang, and Thomas Jensen. "Securing Compilation Against Memory Probing". In: *Proceedings of the 13th Workshop on Programming Languages and Analysis for Security*. PLAS '18. Toronto, Canada: Association for Computing Machinery, 2018, 29–40. ISBN: 9781450359931. DOI: 10.1145/3264820.3264822. URL: https://doi.org/10.1145/3264820.3264822.

[BDJ19] Frédéric Besson, Alexandre Dang, and Thomas Jensen. "Information-Flow Preservation in Compiler Optimisations". In: *CSF 2019 - 32nd IEEE Computer Security Foundations Symposium*. Hoboken, United States: IEEE, June 2019, pp. 1–13. URL: https://hal.inria.fr/hal-02180303.

[BE+04] Hagai Bar-El, Hamid Choukri, David Naccache, Michael Tunstall, and Claire Whelan. "The Sorcerer's Apprentice Guide to Fault Attacks." In: *IACR Cryptology ePrint Archive* 2004 (2004), p. 100. URL: http://dblp.uni-trier.de/db/journals/iacr/iacr2004.html#Bar-ElCNTW04.

[BR10] Gogul Balakrishnan and Thomas Reps. "WYSINWYX: What You See is Not What You eXecute". In: *ACM Trans. Program. Lang. Syst.* 32.6 (Aug. 2010), 23:1–23:84. ISSN: 0164-0925. DOI: 10.1145/1749608.1749612. URL: http://doi.acm.org/10.1145/1749608.1749612.

[Bré+19] Jean-Baptiste Bréjon, Karine Heydemann, Emmanuelle Encrenaz, Quentin Meunier, and Son Tuan Vu. "Fault attack vulnerability assessment of binary code". In: *6th Workshop on Cryptography and Security in Computing*

Systems (CS2). Valencia, Italy, Jan. 2019. ISBN: 978-1-4503-6182-8/19/01. DOI: 10.1145/3304080.3304083.

[Cor+08] Ricardo Corin, Pierre-Malo Deniélou, Cedric Fournet, Karthikeyan Bhargavan, and James Leifer. "A secure compiler for session abstractions". In: *Journal of Computer Security* 16 (Oct. 2008), pp. 573–636. DOI: 10.3233/JCS-2008-0334.

[CS10] Michael R. Clarkson and Fred B. Schneider. "Hyperproperties". In: *J. Comput. Secur.* 18.6 (Sept. 2010), 1157–1210. ISSN: 0926-227X.

[Cuo+12] Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. "Frama-C: A Software Analysis Perspective". In: *10th International Conference on Software Engineering and Formal Methods*. Thessaloniki, Greece, 2012, pp. 233–247. ISBN: 978-3-642-33825-0. DOI: 10.1007/978-3-642-33826-7_16. URL: http://dx.doi.org/10.1007/978-3-642-33826-7_16.

[Cyt+91] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. "Efficiently Computing Static Single Assignment Form and the Control Dependence Graph". In: *ACM Trans. Program. Lang. Syst.* 13.4 (Oct. 1991), 451–490. ISSN: 0164-0925. DOI: 10.1145/115372.115320. URL: https://doi.org/10.1145/115372.115320.

[DPS15] Vijay D'Silva, Mathias Payer, and Dawn Song. "The Correctness-Security Gap in Compiler Optimization". In: *2015 IEEE Security and Privacy Workshops*. 2015, pp. 73–87.

[Dur+16] Louis Dureuil, Guillaume Petiot, Marie-Laure Potet, Thanh-Ha Le, Aude Crohen, and Philippe de Choudens. "FISSC: A Fault Injection and Simulation Secure Collection". In: Sept. 2016, pp. 3–11. ISBN: 978-3-319-45476-4. DOI: 10.1007/978-3-319-45477-1_1.

[Eag07] Michael Eager. "Introduction to the DWARF Debugging Format". In: 2007.

[Ede+16] Kerstin Eder, John P. Gallagher, Pedro López-García, Henk Muller, Zorana Banković, Kyriakos Georgiou, Rémy Haemmerlé, Manuel V. Hermenegildo, Bishoksan Kafle, Steve Kerrison, Maja Kirkeby, Maximiliano Klemen, Xueliang Li, Umer Liqat, Jeremy Morse, Morten Rhiger, and Mads Rosendahl. "ENTRA". In: *Microprocess. Microsyst.* 47.PB (Nov. 2016), pp. 278–286. ISSN: 0141-9331. DOI: 10.1016/j.micpro.2016.07.003. URL: https://doi.org/10.1016/j.micpro.2016.07.003.

[EEA98] Jakob Engblom, Andreas Ermedahl, and Peter Altenbernd. "Facilitating worst-case execution times analysis for optimized code". In: *Proceeding. 10th EUROMICRO Workshop on Real-Time Systems (Cat. No.98EX168)*. 1998, pp. 146–153. DOI: 10.1109/EMWRTS.1998.685079.

[EW14] Hassan Eldib and Chao Wang. "Synthesis of Masking Countermeasures Against Side Channel Attacks". In: *Proceedings of the 16th International Conference on Computer Aided Verification - Volume 8559*. Berlin, Heidelberg: Springer-Verlag, 2014, pp. 114–130. ISBN: 978-3-319-08866-2. DOI: 10.1007/978-3-319-08867-9_8. URL: https://doi.org/10.1007/978-3-319-08867-9_8.

[GM82] Joseph A. Goguen and Jose Meseguer. "Security Policies and Security Models". In: *1982 IEEE Symposium on Security and Privacy* (1982), pp. 11–11.

[Gou+15] Lucien Goubet, Karine Heydemann, Emmanuelle Encrenaz, and Ronald De Keulenaer. "Efficient Design and Evaluation of Countermeasures against Fault Attack with Formal Verification". In: *14th International conference Smart Card Research and Advanced Applications (CARDIS)*. Vol. 9514. Lecture Notes in Computer Science. Bochum, Germany: Springer International Publishing, Nov. 2015, pp. 177–192. DOI: 10.1007/978-3-319-31271-2_11. URL: https://hal.archives-ouvertes.fr/hal-01220291.

[GW+17] Thomas Given-Wilson, Nisrine Jafri, Jean-Louis Lanet, and Axel Legay. "An Automated Formal Process for Detecting Fault Injection Vulnerabilities in Binaries and Case Study on PRESENT". In: *2017 IEEE Trustcom/BigDataSE/ICESS*. 2017, pp. 293–300. DOI: 10.1109/Trustcom/BigDataSE/ICESS.2017.250.

[Hil14] Christoph Hillebold. "Compiler-Assisted Integrits against Fault injection Attacks". MA thesis. University of Technology, Graz, 2014. URL: http://chille.at/articles/master-thesis.

[HLB19] Karine Heydemann, Jean-François Lalande, and Pascal Berthomé. "Formally verified software countermeasures for control-flow integrity of smart card C code". In: *Computers and Security* 85 (Aug. 2019), pp. 202–224. DOI: 10.1016/j.cose.2019.05.004. URL: https://hal.sorbonne-universite.fr/hal-02123836.

[HOM06] Christoph Herbst, Elisabeth Oswald, and Stefan Mangard. "An AES Smart Card Implementation Resistant to Power Analysis Attacks". In: *Proceedings of the 4th International Conference on Applied Cryptography and Network Security*. ACNS'06. Singapore: Springer-Verlag, 2006, pp. 239–252. ISBN: 3-540-34703-8, 978-3-540-34703-3. DOI: 10.1007/11767480_16. URL: http://dx.doi.org/10.1007/11767480_16.

[ISW03] Yuval Ishai, Amit Sahai, and David Wagner. "Private Circuits: Securing Hardware against Probing Attacks". In: *Advances in Cryptology - CRYPTO 2003*. Ed. by Dan Boneh. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 463–481. ISBN: 978-3-540-45146-4.

[Jug+17] Yannis Juglaret, Catalin Hritcu, Arthur Azevedo de Amorim, Boris Eng, and Benjamin C. Pierce. *Beyond Good and Evil: Formalizing the Security Guarantees of Compartmentalizing Compilation*. 2017. arXiv: 1602.04503 [cs.CR].

[Kau+16] 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: Nov. 2016, pp. 573–582. ISBN: 978-3-319-48964-3. DOI: 10.1007/978-3-319-48965-0_36.

[Kir03] Raimund Kirner. "Extending Optimising Compilation to Support Worst-Case Execution Time Analysis". PhD thesis. Technical University of Vienna, May 2003.

[KMW17] Martin S. Kelly, Keith Mayes, and John F. Walker. "Characterising a CPU fault attack model via run-time data analysis". In: *2017 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. 2017, pp. 79–84. DOI: 10.1109/HST.2017.7951802.

[KP01] Raimund Kirner and Peter Puschner. "Transformation of path information for WCET analysis during compilation". In: *Proceedings 13th Euromicro Conference on Real-Time Systems*. 2001, pp. 29–36. DOI: 10.1109/EMRTS.2001.933993.

[LA04] Chris Lattner and Vikram Adve. "LLVM: a compilation framework for lifelong program analysis amp; transformation". In: *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* 2004, pp. 75–86. DOI: 10.1109/CGO.2004.1281665.

[Lat+20] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. *MLIR: A Compiler Infrastructure for the End of Moore's Law*. 2020. arXiv: 2002.11054 [cs.PL].

[Lau+19] Johan Laurent, Christophe Deleuze, Vincent Beroulle, and Florian Pebay-Peyroula. "Analyzing Software Security Against Complex Fault Models with Frama-C Value Analysis". In: *2019 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*. Atlanta, United States: IEEE, Aug. 2019, pp. 33–40. DOI: 10.1109/FDTC.2019.00013. URL: https://hal.univ-grenoble-alpes.fr/hal-02426133.

[Ler06] Xavier Leroy. "Formal certification of a compiler back-end, or: programming a compiler with a proof assistant". In: *POPL 2006: 33rd symposium Principles of Programming Languages*. ACM, 2006, pp. 42–54. DOI: 10.1145/1111037.1111042.

[Ler09] Xavier Leroy. "A Formally Verified Compiler Back-end". In: *J. Autom Reasoning* 43.363 (2009). https://doi.org/10.1007/s10817-009-9155-4.

[Lev07] Ilya Levin. *A byte-oriented AES-256 implementation*. 2007. URL: http://www.literatecode.com/aes256 (visited on 08/14/2019).

[LHB14] Jean-François Lalande, Karine Heydemann, and Pascal Berthomé. "Software countermeasures for control flow integrity of smart card C codes". In: *ESORICS - 19th European Symposium on Research in Computer Security*. Ed. by Miroslaw Kutylowski and Jaideep Vaidya. Vol. 8713. Lecture Notes in Computer Science. Wroclaw, Poland: Springer International Publishing, Sept. 2014, pp. 200–218. DOI: 10.1007/978-3-319-11212-1_12. URL: https://hal.inria.fr/hal-01059201.

[LM97] Yau-Tsun S. Li and Sharad Malik. "Performance analysis of embedded software using implicit path enumeration". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 16.12 (1997), pp. 1477–1487. ISSN: 0278-0070. DOI: 10.1109/43.664229.

[LPR14] Hanbing Li, Isabelle Puaut, and Erven Rohou. "Traceability of Flow Information: Reconciling Compiler Optimizations and WCET Estimation". In: *Proceedings of the 22Nd International Conference on Real-Time Networks and Systems*. RTNS '14. Versaille, France: ACM, 2014, 97:97–97:106. ISBN: 978-1-4503-2727-5. DOI: 10.1145/2659787.2659805. URL: http://doi.acm.org/10.1145/2659787.2659805.

[MB08] Leonardo de Moura and Nikolaj Bjørner. "Z3: an efficient SMT solver". In: vol. 4963. Apr. 2008, pp. 337–340. DOI: 10.1007/978-3-540-78800-3_24.

[MM69] Zohar Manna and John McCarthy. “Properties of Programs and Partial Function Logic”. In: 1969.

[Mor+13] Nicolas Moro, Amine Dehbaoui, Karine Heydemann, Bruno Robisson, and Emmanuelle Encrenaz. “Electromagnetic Fault Injection: Towards a Fault Model on a 32-bit Microcontroller”. In: *2013 Workshop on Fault Diagnosis and Tolerance in Cryptography*. 2013, pp. 77–88. DOI: 10.1109/FDTC.2013.9.

[Mor+14] Nicolas Moro, Karine Heydemann, Emmanuelle Encrenaz, and Bruno Robisson. “Formal verification of a software countermeasure against instruction skip attacks”. In: *Journal of Cryptographic Engineering* 4.3 (Sept. 2014), pp. 145–156. DOI: 10.1007/s13389-014-0077-7. URL: https://hal-emse.ccsd.cnrs.fr/emse-00951386.

[NTZ13] Kedar S. Namjoshi, Giacomo Tagliabue, and Lenore D. Zuck. “A Witnessing Compiler: A Proof of Concept”. In: *Runtime Verification*. Ed. by Axel Legay and Saddek Bensalem. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 340–345. ISBN: 978-3-642-40787-1.

[NZ13] Kedar S. Namjoshi and Lenore D. Zuck. “Witnessing Program Transformations”. In: *Static Analysis*. Ed. by Francesco Logozzo and Manuel Fähndrich. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 304–323. ISBN: 978-3-642-38856-9.

[PAC19] Marco Patrignani, Amal Ahmed, and Dave Clarke. “Formal Approaches to Secure Compilation: A Survey of Fully Abstract Compilation and Related Work”. In: *ACM Comput. Surv.* 51.6 (Feb. 2019). ISSN: 0360-0300. DOI: 10.1145/3280984. URL: https://doi.org/10.1145/3280984.

[Pat+15] Marco Patrignani, Pieter Agten, Raoul Strackx, Bart Jacobs, Dave Clarke, and Frank Piessens. “Secure Compilation to Protected Module Architectures”. In: *ACM Trans. Program. Lang. Syst.* 37.2 (Apr. 2015). ISSN: 0164-0925. DOI: 10.1145/2699503. URL: https://doi.org/10.1145/2699503.

[Per14] Colin Percival. *How to zero a buffer*. 2014. URL: http://www.daemonology.net/blog/2014-09-04-how-to-zero-a-buffer.html (visited on 07/14/2019).

[PG17] Marco Patrignani and Deepak Garg. “Secure Compilation and Hyperproperty Preservation”. In: *2017 IEEE 30th Computer Security Foundations Symposium (CSF)*. 2017, pp. 392–404. DOI: 10.1109/CSF.2017.13.

[PG19] Marco Patrignani and Deepak Garg. *Robustly Safe Compilation or, Efficient, Provably Secure Compilation*. 2019. arXiv: 1804.00489 [cs.PL].

[Pro+17] Julien Proy, Karine Heydemann, Alexandre Berzati, and Albert Cohen. “Compiler-Assisted Loop Hardening Against Fault Attacks”. In: *ACM Trans. Archit. Code Optim.* 14.4 (Dec. 2017), 36:1–36:25. ISSN: 1544-3566. DOI: 10.1145/3141234. URL: http://doi.acm.org/10.1145/3141234.

[Rig+18] Manuel Rigger, Stefan Marr, Stephen Kell, David Leopoldseder, and Hanspeter Mössenböck. “An Analysis of X86-64 Inline Assembly in C Programs”. In: *SIGPLAN Not.* 53.3 (Mar. 2018), 84–99. ISSN: 0362-1340. DOI: 10.1145/3296975.3186418. URL: https://doi.org/10.1145/3296975.3186418.

[RP10] Matthieu Rivain and Emmanuel Prouff. "Provably Secure Higher-Order Masking of AES". In: *Cryptographic Hardware and Embedded Systems, CHES 2010*. Ed. by Stefan Mangard and François-Xavier Standaert. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 413–427. ISBN: 978-3-642-15031-9.

[SC09] Richard M. Stallman and GCC Developer Community. *Using The Gnu Compiler Collection: A Gnu Manual For Gcc Version 4.3.3*. Paramount, CA: CreateSpace, 2009. ISBN: 144141276X, 9781441412768.

[SCA18] Laurent Simon, David Chisnall, and Ross Anderson. "What You Get is What You C: Controlling Side Effects in Mainstream C Compilers". In: *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*. Apr. 2018, pp. 1–15. DOI: 10.1109/EuroSP.2018.00009.

[Sch+18] Bernhard Schommer, Christoph Cullmann, Gernot Gebhard, Xavier Leroy, Michael Schmidt, and Simon Wegener. "Embedded Program Annotations for WCET Analysis". In: *WCET 2018: 18th International Workshop on Worst-Case Execution Time Analysis*. Vol. 63. Barcelona, Spain: Dagstuhl Publishing, July 2018. DOI: 10.4230/OASIcs.WCET.2018.8. URL: https://hal.inria.fr/hal-01848686.

[Sho+16] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. "SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis". In: (2016).

[TSW16] Niek Timmers, Albert Spruyt, and Marc Witteman. "Controlling PC on ARM Using Fault Injection". In: *2016 Workshop on Fault Diagnosis and Tolerance in Cryptography, FDTC 2016, Santa Barbara, CA, USA, August 16, 2016*. 2016, pp. 25–35. DOI: 10.1109/FDTC.2016.18. URL: http://dx.doi.org/10.1109/FDTC.2016.18.

[Wit18] Marc Witteman. *Secure Application Programming in the Presence of Side Channel Attacks*. Tech. rep. 2018.

[Yan+17] Zhaomo Yang, Brian Johannesmeyer, Anders Trier Olesen, Sorin Lerner, and Kirill Levchenko. "Dead Store Elimination (Still) Considered Harmful". In: *Proceedings of the 26th USENIX Conference on Security Symposium*. SEC'17. Vancouver, BC, Canada: USENIX Association, 2017, 1025–1040. ISBN: 9781931971409.

[YSW18] Bilgiday Yuce, Patrick Schaumont, and Marc Witteman. "Fault Attacks on Secure Embedded Software: Threats, Design, and Evaluation". In: *Journal of Hardware and Systems Security* 2.2 (2018), pp. 111–130. ISSN: 2509-3436. DOI: 10.1007/s41635-018-0038-1. URL: https://doi.org/10.1007/s41635-018-0038-1.

[Agt+12] Pieter Agten, Raoul Strackx, Bart Jacobs, and Frank Piessens. "Secure Compilation to Modern Processors". In: *2012 IEEE 25th Computer Security Foundations Symposium*. 2012, pp. 171–185. DOI: 10.1109/CSF.2012.12.

[Com17] Common Criteria. *Common Methodology for Information Technology Security Evaluation, Version 3.1, revision 5*. https://www.commoncriteriaportal.org/. 2017.

[DWA17] DWARF Debugging Information Format Commitee. *DWARF Debugging Information Format Version 5*. 2017. URL: https://dwarfstd.org/doc/DWARF5.pdf.

[Eli19] Eli Bendersky. *pyelftools - Python library for parsing ELF files and DWARF debugging information*. Version 0.26. Dec. 5, 2019. URL: https://github.com/eliben/pyelftools.

[ISO11] ISO C11 Standard. *C11 Standard*. ISO/IEC 9899:2011. 2011. URL: /bib/iso/C11/n1570.pdf.

[LLV20] LLVMdev. *Side-channel resistant values*. 2020. URL: http://lists.llvm.org/pipermail/llvm-dev/2019-September/135079.html (visited on 09/12/2020).

[Lam77] Leslie Lamport. "Proving the Correctness of Multiprocess Programs". In: *IEEE Transactions on Software Engineering* SE-3.2 (1977), pp. 125–143. DOI: 10.1109/TSE.1977.229904.

[The03] The OpenSSL Project. "OpenSSL: The Open Source toolkit for SSL/TLS". www.openssl.org. 2003.