Decision procedures for vulnerability analysis

Benjamin Farinier

Université Grenoble Alpes

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ GRENOBLE ALPES

Spécialité : Mathématiques et Informatique

Arrêté ministériel : 25 mai 2016

Présentée par

Benjamin FARINIER

Thèse dirigée par **Marie-Laure POTET**
et codirigée par **Sébastien BARDIN**

préparée au sein du **Laboratoire VERIMAG**
dans **l'École Doctorale Mathématiques, Sciences et Technologies de l'Information, Informatique**

Procédures de décision pour l'analyse de vulnérabilités

Decision Procedures for Vulnerability Analysis

Thèse soutenue publiquement le **24 juin 2020**,
devant le jury composé de :

Madame Marie-Laure POTET
Professeure des universités, Grenoble INP, Directrice de thèse
Monsieur Sébastien BARDIN
Chercheur, CEA LIST, Codirecteur de thèse
Monsieur Thomas JENSEN
Directeur de recherche, INRIA, Rapporteur
Monsieur Sylvain CONCHON
Professeur des universités, Université Paris-Saclay, Rapporteur
Madame Mihaela SIGHIREANU
Maître de conférences, Université Paris Diderot, Examinatrice
Monsieur Jean GOUBAULT-LARRECQ
Professeur des universités, ENS Paris-Saclay, Examinateur
Monsieur Roland GROZ
Professeur des universités, Grenoble INP, Président du jury

Procédures de décision pour l'analyse de vulnérabilités

Decision Procedures for Vulnerability Analysis

thèse présentée par Benjamin Farinier
en vue de l'obtention du grade de docteur de l'Université Grenoble Alpes

soutenue publiquement le 24 juin 2020

Remerciements

Je remercie tout d'abord Marie-Laure Potet et Sébastien Bardin pour avoir encadré avec tant de patience mes travaux au fil de cette thèse. Je remercie les membres du jury, Mihaela Sighireanu, Jean Goubault-Larrecq, Roland Groz, et en particulier Thomas Jensen et Sylvain Conchon pour leur relecture attentive. Je remercie Claude Marché et les membres de l'équipe VALS pour avoir accepté de m'héberger le temps d'un ATER. Et je remercie tous les membres du LSL, passés, présents et futurs, que j'ai eu grand plaisir à côtoyer, et tout spécialement mes compagnons de galère, les doctorants de l'équipe Binsec.

Je remercie toute ma famille, mes parents, mon frère et mes grands-parents pour leur indéfectible soutien. Je remercie Valentine, Aurélien et Armaël, tous mes amis de Toulouse, de Clermont-Ferrand, de Lyon, de Paris ou d'ailleurs, tous ceux auxquels j'ai pensé sans les nommer, mais qui m'ont accompagné de près ou de loin ces dernières années. Je remercie tout particulièrement Félix et Pauline pour nos traditionnels cafés place d'Aligre, Arnaud pour nos nombreux moments partagés de Lyon à Paris, et Sophie pour nos merveilleuses après-midi thé et pâtisseries, et pour avoir relu tous mes écrits, cette thèse y compris.

Enfin je remercie Claire pour avoir su être là au cours de toutes ces années, et dont la présence m'a permis de mener à bien cette thèse.

Résumé

L'Exécution symbolique est une technique de vérification formelle qui consiste en modéliser les exécutions d'un programme par des formules logiques pour montrer que ces exécutions vérifient une propriété donnée. Très efficace, l'Exécution symbolique a permis le développement d'outils d'analyse à l'origine de la découverte de nombreux bogues. Il est question aujourd'hui de l'employer dans d'autres contextes que la recherche de bogues, comme en analyse de vulnérabilités. L'application de l'Exécution symbolique à l'analyse de vulnérabilités diffère de la recherche de bogues sur au moins deux aspects :

- Premièrement, trouver un bogue n'implique pas avoir trouvé une vulnérabilité, et faire la différence entre les deux requiert passer d'une analyse du code source à une analyse du code binaire. Or les sémantiques utilisées en analyse de code binaire sont non seulement moins structurées que celles utilisées en analyse de code source, mais aussi bien plus verbeuses.

- Deuxièmement, les interactions avec l'environnement ne se modélisent pas de la même manière en recherche de bogues qu'en analyse de vulnérabilités, pour laquelle la modélisation de l'environnement va dépendre du modèle d'attaquant choisi. En effet, pour conclure qu'une vulnérabilité en est une, il faut montrer qu'elle se manifeste dans toutes les configurations que peuvent prendre les composantes de l'environnement non contrôlées par l'attaquant.

Ces deux différences vont avoir un impact profond sur les formules logiques générées par l'Exécution symbolique et leur résolution :

- Les formules logiques générées au cours de l'Exécution symbolique deviennent rapidement gigantesques et de plus en plus difficiles à résoudre. Si ce problème n'est pas spécifique à l'analyse de vulnérabilités, il se renforce dans ce contexte en raison de la verbosité des sémantiques utilisées en analyse de code binaire.

- La modélisation de certaines propriétés de sécurité est susceptible de faire intervenir des quantificateurs dont l'emploi rend les formules logiques générées presque impossibles à résoudre. Il en va ainsi en analyse de vulnérabilités, selon qu'un attaquant contrôle ou non un composant de l'environnement.

Cette thèse porte donc sur deux problématiques issues du domaine des procédures de décision, à savoir la simplification de formules logiques afin de limiter l'explosion de leur taille au cours de l'analyse, et l'extension au cas quantifié de la logique des solveurs afin de permettre des modélisations plus fines, nécessaires à l'analyse de vulnérabilités.

Abstract

Symbolic Execution is a formal verification technique which consists in modeling program executions by logical formulas in order to prove that these executions verify a given property. Very effective, Symbolic Execution led to the development of analysis tools and to the discovery of many new bugs. The question is now how to use it in other contexts than bug finding, for example in vulnerability analysis. Applying Symbolic Execution to vulnerability analysis fundamentally differs from bug finding on at least two aspects:

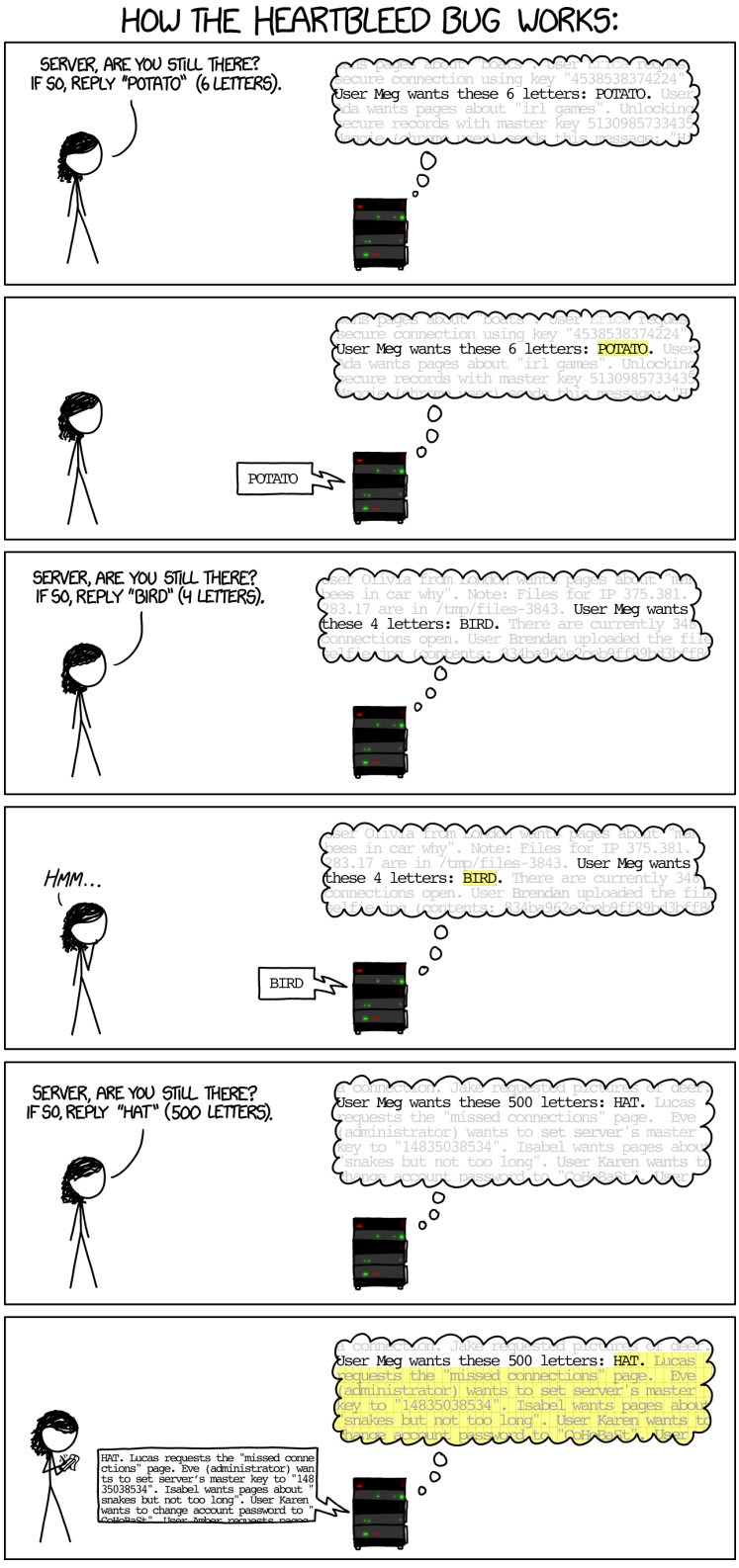
- First, finding a bug does not imply having found a vulnerability, and differentiating between the two requires to move from source-level analysis to binary-level analysis. But binary-level semantics are not only less structured than source-level semantics, they are also far more verbose.
- Second, interactions with the environment are not modeled in the same way for vulnerability analysis than for bug finding. Indeed in vulnerability analysis, the environment model will depend on the attacker model we choose, as to conclude that a vulnerability is a real one, we need to prove the vulnerability manifests itself for all the configurations that components not controlled by the attacker can take.

These two differences have a profound impact on logical formulas generated by Symbolic Execution and their resolution:

- Logical formulas generated during an analysis quickly become gigantic, and more and more difficult to solve. If this problem is not specific to vulnerability analysis, it is reinforced in this context, because of the verbosity of semantics used in binary-level analysis.
- Modeling some security properties is likely to involve quantifiers whose use made generated logical formulas nearly impossible to solve. This is the case in vulnerability analysis about interactions with the environment, according to whether an attacker does or does not control a component.

Therefore this thesis focuses on two issues arising from the field of decision procedures, namely the simplification of logical formulas to limit their size explosion during the analysis, and the extension of solvers to quantified logic in order to allow finer models, required for vulnerability analysis.

Contents



Randall Munroe, *Heartbleed Explanation*, https://xkcd.com/1354/

Préface

Le 7 avril 2014, la vulnérabilité logicielle Heartbleed [hea14] est rendue publique. Au moment de sa découverte, entre 24% et 55% des serveurs dits sécurisés seraient touchés [DKA+14], soit environ un demi-million de serveurs, ce qui compromet une part significative des échanges sur Internet. Cette faille de sécurité est due à un Bogue introduit par erreur le 14 mars 2012 dans la bibliothèque de cryptographie open source OpenSSL, une implémentation largement déployée des protocoles SSL/TLS de sécurisation des échanges sur Internet. Elle permet à un attaquant de lire la mémoire d'un serveur vulnérable, et donc sous certaines conditions de récupérer les clefs secrètes de chiffrement, les identifiants et mots de passe des utilisateurs, ou les contenus échangés. Heureusement, les conséquences de cette vulnérabilité resteront limitées en dépit de sa gravité, principalement grâce à la promptitude de la communauté informatique à déployer le correctif qui permit d'éviter une utilisation de grande ampleur de Heartbleed.

Le plus surprenant avec Heartbleed est peut être le décalage entre la dangerosité de la vulnérabilité et la simplicité de sa cause. En effet, cette faille venait du fait qu'un utilisateur pouvait demander à lire un certain nombre de caractères dans une chaîne de caractères, sans qu'il soit vérifié que la chaîne en question contienne suffisamment de caractères. Dans le cas où le nombre demandé de caractères était plus grand que la taille de la chaîne, les caractères excédants étaient lus dans les données consécutives à la chaîne en mémoire, données normalement inaccessibles. Ce type de vulnérabilité, dit par *dépassement de tampon* (*buffer overflow* en anglais), était déjà connue en 1972 [And72], et faisait partie des failles exploitées par le ver informatique Morris pour se répandre en 1988. Le fait que des causes connues de longue date comme étant dangereuses soient aujourd'hui encore à la source de failles de sécurité majeures démontre la nécessité et la pertinence de la recherche en vérification formelle de programmes.

Vérification formelle de programmes

La vérification formelle [CW96] vise à prouver ou réfuter la correction d'un système vis-à-vis d'une spécification formelle. Cette spécification est exprimée comme un ensemble de propriétés formalisées en logique mathématique dont on va prouver la validité pour un modèle donné du système. Dans le cas de la vérification de programme, les systèmes étudiés sont des programmes dont les modèles sont obtenus par la sémantique de leur code source. Les propriétés vérifiées sont, elles, d'un registre très large qui va de la preuve d'absence d'erreurs à l'exécution, ou de l'intégrité mémoire du programme, à la preuve de correspondance entre un algorithme et son implémentation.

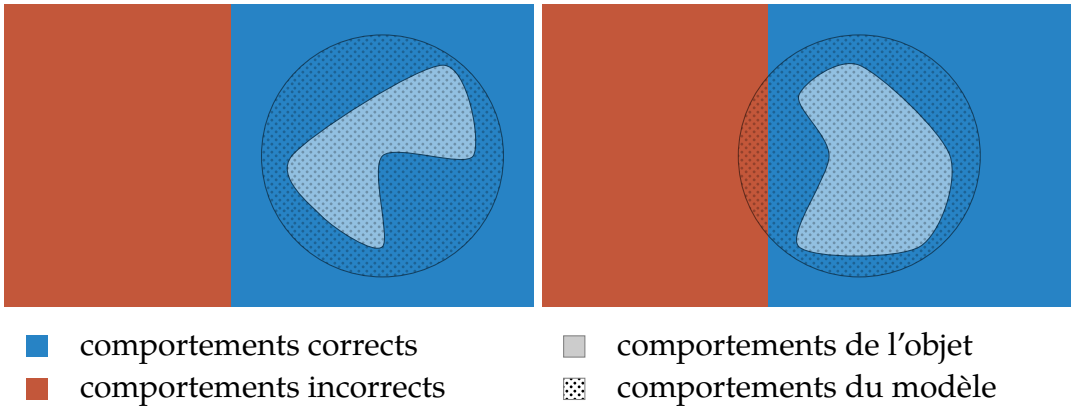


FIGURE 1 – Illustration d'une sur-approximation, qui conclut à la correction de l'objet vis-à-vis de sa spécification dans la figure de gauche, mais ne parvient à aucune conclusion du fait de *faux positifs* dans la figure de droite.

Sur-approximation L'idée de vérifier formellement des composants cruciaux en terme de sécurité n'est pas nouvelle, que ce soit au niveau des protocoles [BBK17] ou de leurs implémentations [KKP+15, BFK16, BBD+17]. Malheureusement, l'usage de bibliothèques logicielles formellement vérifiées reste peu répandu, leur développement étant significativement plus complexes, tandis que celles existantes ne se prêtent que difficilement à la vérification à posteriori. En effet, la plupart des outils de vérification formelle travaille sur des modèles qui sur-approximent les comportements des objets étudiés [Hoa69, CC77] : tous les comportements d'un l'objet sont capturés par son modèle, mais un modèle considère des comportements qui ne sont pas réalisables par son objet, comme illustré dans la Figure 1. Ainsi, lorsque l'outil conclut à la correction du modèle vis-à-vis de la spécification, alors l'objet l'est aussi car l'ensemble de ses

comportements est inclus dans l'ensemble de ceux qui ont été vérifiés. Par contre, lorsque l'outil conclut à l'incorrection du modèle vis-à-vis de la spécification, le comportement coupable de l'incorrection peut être propre au modèle et l'objet être, lui, correct vis-à-vis de la spécification. On parle alors de *faux positif*. Or lorsqu'un programme n'a pas été développé avec l'objectif d'être ensuite vérifié formellement, sa modélisation par l'outil sera tellement sur-approximée qu'il sera dans presque tous les cas impossible de conclure à la correction du programme. Et si le programme est réellement incorrect, les véritables erreurs seront noyées dans le trop grand nombre de faux positifs.

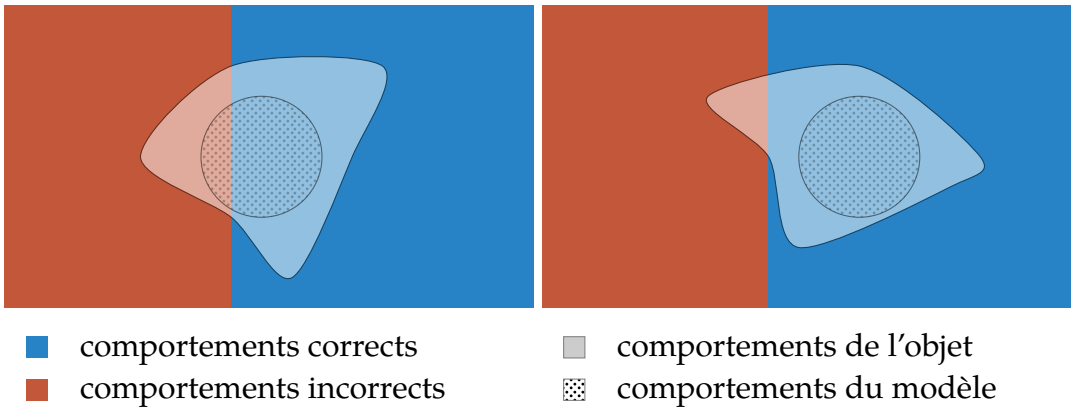


FIGURE 2 – Illustration d'une sous-approximation, qui conclut à l'incorrection de l'objet vis-à-vis de sa spécification dans la figure de gauche, mais ne parvient à aucune conclusion du fait de *faux négatifs* dans la figure de droite.

Sous-approximation Cependant, bien qu'elles soient moins fréquentes, certaines techniques de vérification formelle fonctionnent par *sous-approximation* [CKL04, CS13] : tous les comportements du modèle sont des comportements de l'objet, mais certains comportements de l'objet manquent au modèle, comme illustré dans la Figure 2. Ici, toutes les incorrections relevées sur le modèle s'appliquent à l'objet, on est donc exempt de faux positif. La contrepartie est que l'outil ne peut conclure à la correction de l'objet, puisque même si le modèle est prouvé correct, certains comportements de l'objet n'auront pas été vérifiés, comportements pouvant contenir des erreurs. Ce sont cette fois des *faux négatifs*. L'intérêt de ces approches par sous-approximation est de permettre d'éliminer a priori de nombreux bogues à bas coût. De plus, ces dernières peuvent être appliquées à des programmes qui n'avaient pas été initialement pensés pour la vérification, dans l'optique d'éliminer un maximum de comportements incorrects et de rendre ces programmes plus aptes à un processus de vérification par sur-approximation.

```
int main () {
  int x = input();
  int y = input();
  int z = 2 * y;
  if (z == x) {
    if (x > y + 10)
      printf("Success!\n");
  }
  printf("Failure...\n");
}
```

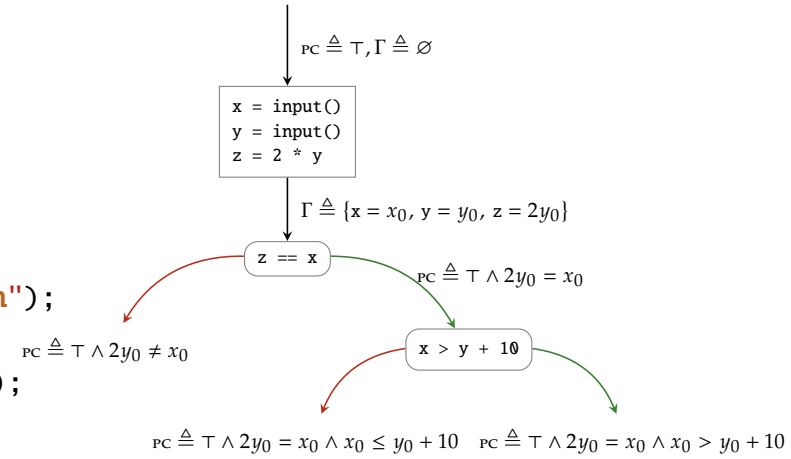


FIGURE 3 – Exécution symbolique d'un petit programme. Le prédicat de chemin PC collecte les contraintes que les entrées doivent satisfaire pour que l'exécution concrète du programme suive un chemin spécifique, tandis que l'état symbolique Γ suit l'évolution de la représentation symbolique des variables du programme.

Exécution symbolique

L'Exécution symbolique fait partie de ces techniques de vérification formelle par sous-approximation [BGM13, CS13]. Intuitivement, la vérification par Exécution symbolique consiste en énumérer les exécutions possibles d'un programme, et déterminer pour chacune d'entre elles une entrée sur laquelle le programme la réalise. La Figure 3 illustre l'Exécution symbolique d'un petit programme. On vérifie ensuite que chacune de ces exécutions satisfait bien la propriété que l'on cherche à vérifier. Si l'une des exécutions énumérées invalide cette propriété, on peut conclure à l'incorrection du programme vis-à-vis de sa spécification. Mais si l'on veut pouvoir conclure à la correction du programme, on doit prouver que toutes ses exécutions possibles satisfont la propriété. Or, dans bien des cas le nombre de ces exécutions est immense, voire infini dans certains cas ; il est donc impossible de toutes les énumérer. L'Exécution symbolique est donc bel et bien une technique de vérification formelle par sous-approximation.

L'Exécution symbolique a été tout d'abord introduite dans les années 1970, mais ce n'est que récemment qu'elle a été redécouverte et rendue utilisable [GKS05, SMA05, WMMR05, CGP+06]. Si l'idée derrière l'Exécution symbolique reste simple, elle s'est révélée être en pratique très efficace pour la recherche de bogues, et a permis le développement d'outils d'analyse de programmes à l'origine de la découverte de nombreux bogues dans des logiciels très usités [CDE08]. Son efficacité est telle qu'il est

aujourd'hui question d'essayer de l'employer dans d'autres contextes que la recherche de bogues, par exemple l'analyse de vulnérabilités. Cependant, l'application de l'Exécution symbolique à l'analyse de vulnérabilités diffère fondamentalement de la recherche de bogues sur au moins deux aspects :

Sémantique de bas niveau vs. sémantique de haut niveau Trouver un bogue n'implique pas avoir trouvé une vulnérabilité, et faire la différence entre les deux requiert bien souvent de passer d'une analyse au niveau du code source à une analyse au niveau du code binaire. Ainsi la recherche de bogues peut être assimilée à la recherche de comportements indéterminés dans la sémantique du code source du programme. Cependant, trouver une vulnérabilité demande en sus de s'assurer que ces comportements indéterminés sont possiblement exploitables, ce qui nécessite de modéliser leurs effets sur l'état global du programme, et donc de passer à une sémantique du programme de niveau binaire. Or ces sémantiques de niveau binaire sont non seulement moins structurées que celles utilisées au niveau du code source, mais elles sont aussi bien plus verbeuses, ce qui les rend significativement plus difficiles à analyser.

Analyse de sécurité vs. analyse de sûreté Les interactions avec l'environnement ne se modélisent pas de la même manière en analyse de vulnérabilités qu'en recherche de bogues. En effet, tout programme qui s'exécute le fait au sein d'un environnement avec lequel il va interagir, et qui doit donc être lui aussi modélisé. En recherche de bogues, il est raisonnable de conclure que le programme contient un bogue s'il existe une configuration de l'environnement dans laquelle l'exécution du programme produit ledit bogue. Mais en analyse de vulnérabilités, la modélisation de l'environnement va dépendre du modèle d'attaquant que l'on choisit. En effet, il est peu raisonnable de supposer que l'attaquant contrôle entièrement l'environnement dans lequel s'exécute le programme, auquel cas il pourrait simplement remplacer le programme par un autre. C'est donc qu'il existe des composantes de l'environnement que l'attaquant ne contrôle pas. Ainsi, pour qu'une vulnérabilité en soit une, il faut qu'elle puisse se manifester pour toutes les configurations que peuvent prendre ses composantes non contrôlées.

En plus de nécessiter le développement d'analyses spécifiques pour l'analyse de vulnérabilités, ces deux différences vont avoir un impact profond sur un des composants cruciaux pour ces techniques, les solveurs SMT.

Procédures de décision

En effet, si nous avons déjà évoqué la modélisation du système à vérifier, nous n'avons pas discuté de la preuve de la validité d'une propriété pour un modèle donné du système. On parle de procédures de décision puisqu'il s'agit de décider de façon procédurale si oui ou non la propriété est vérifiée [KS08]. Le plus souvent, le modèle du système et la formalisation de la propriété à vérifier sont tous deux exprimés dans une même logique mathématique, dans laquelle on va prouver que le premier implique le second. C'est cette preuve qui, selon l'expressivité de la logique choisie, va faire l'objet d'une automatisation plus ou moins importante. En effet, plus une logique est expressive, et plus il est difficile d'en automatiser les raisonnements. Par exemple, dans un assistant de preuve interactif, les logiques utilisées sont particulièrement expressives [CH88] : il est impossible d'en automatiser le raisonnement ; c'est donc à l'utilisateur de construire la preuve, l'assistant se chargeant alors de vérifier que la preuve est correcte. Inversement, en Exécution symbolique, la propriété à vérifier doit l'être sur chaque trace d'exécution, résultant en un nombre d'obligations de preuves bien trop grand pour ne pas nécessiter une automatisation complète : les logiques utilisées seront donc bien moins expressives.

La logique la plus fréquemment rencontrée en Exécution symbolique est la logique du premier ordre sans quantificateurs, avec l'égalité et combinées à des théories dans lesquelles sont exprimées certains symboles. Des exemples de théories incluent l'algèbre des booléens, l'arithmétique linéaire pour les entiers et les réels, la théorie des tableaux avec extensionnalité, ou celle des vecteurs de bits de taille fixe. Les solveurs pour cette logique combinent des procédures de décisions pour les différentes théories afin de savoir si pour une formule donnée il existe une solution qui la satisfasse. On parle donc de solveurs pour le problème de décision de la Satisfiabilité Modulo Théorie [BSST09, BST10], de solveurs SMT.

L'utilisation de solveurs SMT induit pour l'Exécution symbolique et son application à l'analyse de vulnérabilités les problématiques suivantes :

Formules de grande taille Les formules logiques générées au cours d'une analyse deviennent rapidement gigantesques. En effet, l'Exécution symbolique fonctionne par déroulement du programme : le corps d'une fonction est copié dans la trace d'exécution à chaque exécution de ladite fonction. Il en va de même de la traduction logique de la fonction dans la formule finale, qui devient de plus en plus difficile à résoudre pour le solveur. Ainsi, l'Exécution symbolique d'un programme de très petite taille contenant une fonction s'exécutant en boucle à l'infini engendrera une séquence de formules de taille croissant à l'infini. Et

si ce problème n'est pas spécifique à l'analyse de vulnérabilités, il se renforce dans ce contexte, car confirmer qu'un bogue est une vulnérabilité nécessite de passer à des sémantiques de niveau binaire extrêmement verbeuses, ce qui a pour conséquence de produire des formules plus grandes encore. Par exemple dans le Section 6.5.4 nous présentons le cas d'étude ASPack, dans lequel l'Exécution symbolique produit d'immenses formules de 96 MB, qui contiennent plus de 363 000 opérations logiques, et dont la résolution par les solveurs prend plus de 24 heures.

Formules quantifiées La logique du premier ordre sans quantificateurs n'est pas suffisamment expressive pour permettre la modélisation de certaines propriétés de sécurité. C'est particulièrement le cas en analyse de vulnérabilités à propos des interactions du programme avec l'environnement. Leur modélisation est susceptible de faire intervenir des quantificateurs, en fonction du modèle d'attaquant choisi, selon par exemple qu'il en contrôle un composant ou non. Par exemple, le fragment de code dans Figure 5.1 contient une variable non-déterministe dont la valeur intervient dans une condition de garde. Comme cette variable est incontrôlable nous aimerions pouvoir dire que les attaquants doivent trouver un moyen de contourner la condition de garde quelle que soit la valeur de la variable, ce qui nécessite des quantificateurs. Ce manque dans la logique va amener des imprécisions dans la modélisation qui pourront autoriser des comportements qui ne sont pas possibles en réalité, et ainsi introduire des faux positifs, là où pourtant l'Exécution symbolique en tant que technique de vérification par sous-approximation ne devrait pas en avoir. Il est à noter que le problème dual se pose avec les méthodes par sur-approximation lorsque l'environnement n'est pas assez général, ne modélise pas certains comportements possibles, et induit donc des faux négatifs.

Ce sont donc sur deux problématiques issues du domaine des procédures de décision que les travaux de cette thèse portent, à savoir la simplification de formules logiques afin de limiter l'explosion de leur taille au cours de l'analyse et de faciliter leur résolution, et l'extension au cas quantifié de la logique des solveurs SMT afin de permettre des modélisations plus fines évitant l'écueil de l'apparition de faux positifs, le tout dans le but d'améliorer l'état de l'art en Exécution symbolique pour l'analyse de vulnérabilités.

Part I

Introduction

Chapter 1

Introduction

On April 7, 2014, the Heartbleed [hea14] software vulnerability is made public. At the time of its discovery, between 24% and 55% of so-called secure servers would have been affected [DKA+14], or about half a million servers, compromising a significant share of Internet exchanges. This security issue is due to a bug mistakenly introduced on March 14, 2012 in the OpenSSL open source cryptography library, a widely deployed implementation of secured Internet exchange protocols SSL/TLS. It allows an attacker to read the memory of a vulnerable server, and therefore under certain conditions to recover secret encryption keys, user IDs and passwords, or exchanged content. Fortunately, despite the severity of this vulnerability, consequences were to be limited, mainly thanks to the readiness of the IT community who quickly deployed the security patch and prevented a large-scale use of Heartbleed.

Perhaps the most surprising thing about Heartbleed is the discrepancy between the danger of the vulnerability and the simplicity of its cause. Indeed, this flaw stems from the fact that a user could ask to read a certain number of characters in a string, without verifying that the string in question contains enough characters. In the case where the requested number of characters was larger than the size of the string, exceeding characters were read in data consecutive to the string in memory, data which should normally be inaccessible. This kind of vulnerability called *buffer overflow* was already known in 1972 [And72], and was one of those exploited by the Morris computer worm to spread in 1988. The fact that long-standing causes known as dangerous are still the source of major security breaches demonstrates the need for and the relevance of research in formal verification.

1.1 Automated Software Verification

Formal verification [CW96] aims to prove or disprove the correctness of a system against a formal specification. This specification is expressed as a set of properties formalized in mathematical logic which will be proved to hold for a given model of the system. In the case of automated software verification, studied systems are programs whose models are obtained from the semantics of their source code. Verified properties are of a large variety, ranging from the absence of runtime errors, or the integrity of the program memory, to the proper realisation of an algorithm by its implementation.

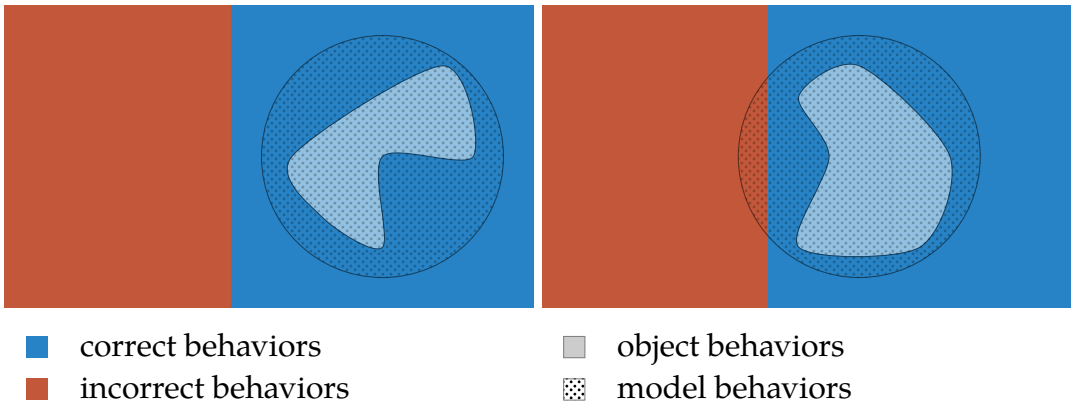


Figure 1.1 – Illustration of an over-approximation, which correctly concludes to the object correctness with regard to its specification in the figure on the left, but fails to conclude because of *false positives* in the figure on the right.

Over-Approximation Formally verifying crucial security components is not a new idea, either at protocols level [BBK17] or at implementation level [KKP+15, BFK16, BBD+17]. Unfortunately, the use of formally verified software libraries remains limited, because developing formally verified libraries is still tremendously complex, while existing non formally verified libraries do not lend themselves easily to a posteriori verification. Indeed, most formal verification tools work on models that *over-approximate* the behavior of studied objects [Hoa69, CC77]: all the possible behaviors of an object are captured by its model, but a model also considers behaviors that are not realizable by the object, as pictured in Figure 1.1. Thus, when over-approximating tools conclude to the model correctness with regard to a specification, then the object is also correct, as all of its behaviors are included in those that have been verified. On the other hand, when over-approximating tools conclude to the model incorrectness, the behavior

responsible for the error may be model specific and the object still be correct with regard to the specification. We are in the presence of *false positives*. But when a program was not developed with the intention of being formally verified, its model will be so over-approximated that it will be almost always impossible to conclude to the program correctness. And if the program is really incorrect, the real mistakes will be drowned in too many false positives.

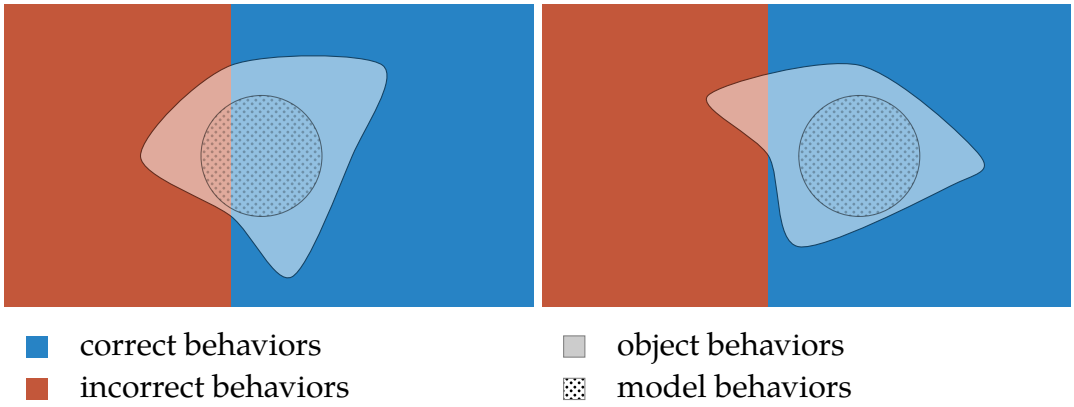


Figure 1.2 – Illustration of an under-approximation, which correctly concludes to the object incorrectness with regard to its specification in the figure on the left, but fails to conclude because of *false negatives* in the figure on the right.

Under-Approximation Although less frequent, some formal verification techniques work by *under-approximation* [CKL04, CS13]: all behaviors of the model are realizable by the object, but some behaviors of the object are missing from those of the model, as pictured in Figure 1.2. Here, all the errors found on the model apply to the object, we are free of false positives. The counterpart is that under-approximating tools cannot conclude to the object correctness, since even if the model is proved to be correct, some behaviors of the object will not have been verified, behaviors that may contain errors. This time we are in the presence of *false negatives*. The interest of these under-approximating approaches is to eliminate in advance lurking corner bugs in a cost effective way. It is also possible to use them on programs not initially thought for formal verification, in order to eliminate a maximum of incorrect behaviors, making these programs more suitable for over-approximating formal verification techniques.

```
int main () {
  int x = input();
  int y = input();
  int z = 2 * y;
  if (z == x) {
    if (x > y + 10)
      printf("Success!\n");
  }
  printf("Failure...\n");
}
```

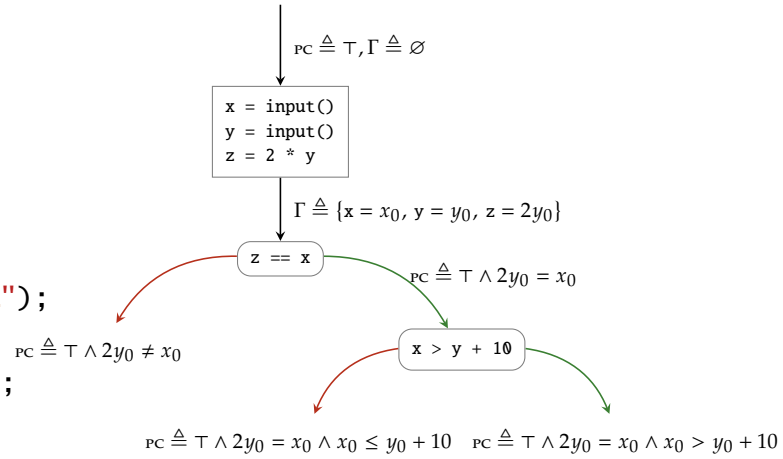


Figure 1.3 – Symbolic Execution of a small program. Path constraint PC gathers constraints that inputs have to satisfy to ensure that the program concrete execution follows a specific path, while symbolic state Γ keeps track of program variables symbolic representations.

1.2 Symbolic Execution

One of these under-approximating formal verification techniques is Symbolic Execution [BGM13, CS13]. Intuitively, Symbolic Execution consists in enumerating all the possible execution paths of a given program and to determine for each path an input on which the program realizes it. Figure 1.3 illustrates the Symbolic Execution of a small program. Then for each one of these paths we check if this execution path satisfies the property we aim to verify. If one of the enumerated execution paths invalidates this property, then we conclude to the program incorrectness with regard to its specification. However if we want to conclude to the program correctness, we must prove that all of its possible execution paths satisfy the property. But in many cases the number of these paths is huge, even sometimes infinite, therefore it is not possible to list them all. Symbolic Execution is indeed an under-approximating formal verification technique.

Symbolic Execution was first introduced in the mid 1970s, but rediscovered and made practical only recently [GKS05, SMA05, WMMR05, CGP+06]. While the idea behind Symbolic Execution remains simple, it has proven to be very effective for bug finding, and led to the development of several program analysis tools and to the discovery of many new bugs in very used softwares [CDE08]. It is in fact so effective that the question is now how to use it in other contexts than bug finding, for example in vulnerabilities analysis. Applying Symbolic Execution to vulnerability analysis

fundamentally differs from bug finding on at least two aspects:

Low-Level Semantics vs. High-Level Semantics Finding a bug does not imply having found a vulnerability, and to differentiate between the two often requires to move from source-level analysis to binary-level analysis. Indeed, bug finding can be seen as looking for undefined behaviors in the program source code semantics. However finding a vulnerability requires in addition to ensure that these undefined behaviors can be exploited, which requires modeling their effects on the global program state, and thus to move on a binary-level semantic of the program. But binary-level semantics are not only less structured than source-level semantics, they are also far more verbose, making them significantly harder to analyze;

Security Analysis vs. Safety Analysis Interactions with the environment are not modeled in the same way for vulnerability analysis and for bug finding. Indeed, any program is executed within an environment with which it will interact, and which must therefore be modeled as well. In bug finding, it is reasonable to conclude the program contains a bug if there is a configuration of the environment for which the program execution leads to that bug. However in vulnerability analysis, the environment model will depend on the attacker model we choose. Indeed, it is unreasonable to assume that the attacker completely controls the environment in which the program runs, because in this case the attacker could basically replace the program with another one. Thus there are environment components that the attacker does not control. Therefore a vulnerability is a real one if it manifests itself for all possible configurations adopted by uncontrolled components.

These two differences not only require the development of dedicated vulnerabilities analysis strategies, they also have a profound impact on crucial components for automated software verification techniques: SMT solvers.

1.3 Decision Procedures

Indeed, if we already mentioned models of the system to check, we did not discuss about validity proofs of a property for a given model of the system. The question here is to decide automatically whether a property holds or not, hence we talk about decision procedures [KS08]. Most of the time, the model of the system and the formalization of the property to be verified are both expressed in the same mathematical logic in which we will prove that the first implies the second. This proof will be subject of more or less automation depending on the expressiveness of the chosen logic. Indeed,

the more expressive a logic is, the more difficult it is to automate the reasoning. For example, logics used in an interactive proof assistant are particularly expressive [CH88]: it is impossible to automate the reasoning, therefore it comes to the user to build the proof, the assistant only ensuring that the proof is correct. Conversely in Symbolic Execution, the property is verified for each execution path, resulting in a number of proof obligations far too great not to require complete automation: we have to use much less expressive logics.

The most frequently encountered logic in Symbolic Execution is the quantifier-free first-order logic with equality and combined with theories expressing certain symbols. Theories of interest include boolean algebrea, real and integer arithmetic, arrays with extensionality, or bit-vectors with arbitrary size. First-order solvers combine decision procedures for the different theories in order to know if for a given formula there exists a solution that satisfies it. We therefore speak of solvers for the decision problem of the Satisfiability Modulo Theory [BSST09, BST10], or SMT solvers.

The use of SMT solvers induces for Symbolic Execution and its application to vulnerability analysis the following challenges:

Large Size Formulas Logical formulas generated during an analysis quickly become gigantic. Indeed, Symbolic Execution works by program unrolling: an execution path contains a copy of a function body for each time the execution goes through this function. The same goes for the logical translation of the execution path into the final formula, which becomes more and more difficult to solve. Thus symbolically executing a very small program containing a function running in loop at infinity will generate a sequence of formulas of size increasing toward infinity. And if this problem is not specific to vulnerability analysis, it is reinforced in this context, because confirming a bug is a vulnerability requires to move on overly verbose binary-level semantics, producing even bigger formulas. For example in Section 6.5.4 we present the ASPack case study where Symbolic Execution produces huge formulas of 96 MB, containing more than 363 000 logical operations, and for which solvers spend more than 24 hours in resolution;

Quantified Formulas The quantifier-free first-order logic is not sufficiently expressive to model some security properties. This is particularly the case in vulnerability analysis about interactions with the environment. Modeling them is likely to involve quantifiers, depending on the chosen attacker model according to for example whether it controls a component or not. For example, the code snippet in Figure 5.1 contains a non-deterministic variable whose value intervene in a guard condition. As this variable is uncontrollable, we would like to say that

attackers have to find a way to bypass the guard condition whatever the value of the variable is, which requires quantifiers. This lack in the logic will lead to imprecise models that may allow behaviors impossible in reality, and thus introduce false positives where Symbolic Execution, as a under-approximating verification technique, should not to have any. Note that the dual problem arises with over-approximating verification techniques when the environment is not general enough, does not model some possible behaviors, and thus induces false negatives.

It is therefore on these two issues arising from the field of decision procedures that this thesis focuses, namely the simplification of logical formulas to limit their size explosion during the analysis and to facilitate their resolution, and the extension of SMT solvers to quantified logic in order to allow finer models avoiding the pitfall of the appearance of false positives, for the purpose of improving the state of the art in Symbolic Execution and vulnerability analysis.

1.4 Contributions and Organization of the Document

Contributions This thesis makes the following contributions:

- We highlight with realistic case studies the false positives issue for classic Symbolic Execution, and especially for Symbolic Execution applied to vulnerability analysis. We show how this issue can be overcome by switching to a low-level semantics and a better modeling of interactions with the environment, but at the price of generating larger formulas involving quantifiers;

- We propose in [FBBP18] a novel and generic taint-based approach for model generation of quantified formula and prove its correctness and its efficiency under reasonable assumptions. We present a concrete implementation of our method specialized on arrays and bit-vectors that we evaluate on SMT-LIB benchmarks and formulas generated by the binary-level Symbolic Execution tool Binsec;

- We present in [FDBL18] a new preprocessing step for scalable and thorough formulas simplification. We experimentally evaluate it in different settings and we show that the technique is fast, scalable, and yields a significant reduction of the formulas size with always a positive impact on resolution time. This impact is even dramatic for formulas generated by Symbolic Execution;

- Finally, we formally define a revision of Symbolic Execution that we called *Robust Symbolic Execution*, and which aims to eliminate the false positives issue. We describe in detail a first implementation of Robust Symbolic Execution in the binary-level Symbolic Execution tool BINSEC, and report some encouraging initial experimental results.

Organization of the Document The rest of this document is organized as follows:

Chapter 2 presents as a motivating example a realistic vulnerability analysis case study which highlights the false positives issue for classic Symbolic Execution;

Chapter 3 introduces the many-sorted first-order logic, the formal system used by Symbolic Execution and other automatic software verification techniques;

Chapter 4 explains what is Symbolic Execution, the automated software analysis technique on which this thesis focuses, and how to adapt it to formal verification;

Chapter 5 describes and evaluates our taint-based approach to the model generation problem for quantified formulas;

Chapter 6 describes and evaluates our preprocessing step for scalable and thorough formulas simplification;

Chapter 7 defines Robust Symbolic Execution, a Symbolic Execution which is really exempt of false positives.

Bibliography

[And72] James P. Anderson. Computer security technology planning study. Technical Report ESD-TR-73-51, Vol.II, Air Force Electronic Systems Division, 1972.

[BBD+17] Benjamin Beurdouche, Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, Pierre-Yves Strub, and Jean Karim Zinzindohoue. A messy state of the union: taming the composite state machines of TLS. *Commun. ACM*, 60(2):99–107, 2017.

[BBK17] Karthikeyan Bhargavan, Bruno Blanchet, and Nadim Kobeissi. Verified models and reference implementations for the TLS 1.3 standard candidate.

In *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*, pages 483–502, 2017.

[BFK16] Karthikeyan Bhargavan, Cédric Fournet, and Markulf Kohlweiss. mitls: Verifying protocol implementations against real-world attacks. *IEEE Security & Privacy*, 14(6):18–25, 2016.

[BGM13] Ella Bounimova, Patrice Godefroid, and David A. Molnar. Billions and billions of constraints: whitebox fuzz testing in production. In *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, pages 122–131, 2013.

[BSST09] Clark W. Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. Satisfiability Modulo Theories. In *Handbook of Satisfiability*, pages 825–885. 2009.

[BST10] Clark Barrett, Aaron Stump, and Cesare Tinelli. The SMT-LIB Standard: Version 2.0. In A. Gupta and D. Kroening, editors, *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, UK)*, 2010.

[CC77] Patrick Cousot and Radia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *4th ACM Symposium on Principles of Programming Languages (POPL)*. ACM, 1977.

[CDE08] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*, pages 209–224, 2008.

[CGP+06] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. EXE: automatically generating inputs of death. In *Proceedings of the 13th ACM Conference on Computer and Communications Security, CCS 2006, Alexandria, VA, USA, Ioctober 30 - November 3, 2006*, pages 322–335, 2006.

[CH88] Thierry Coquand and Gérard P. Huet. The calculus of constructions. *Inf. Comput.*, 76(2/3):95–120, 1988.

[CKL04] Edmund M. Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In *TACAS, Barcelona, Spain, March 29 - April 2, 2004*, pages 168–176, 2004.

[CS13] Cristian Cadar and Koushik Sen. Symbolic execution for software testing: three decades later. *Commun. ACM*, 56(2):82–90, 2013.

[CW96] Edmund M. Clarke and Jeannette M. Wing. Formal methods: State of the art and future directions. *ACM Comput. Surv.*, 28(4):626–643, 1996.

[DKA+14] Zakir Durumeric, James Kasten, David Adrian, J. Alex Halderman, Michael Bailey, Frank Li, Nicholas Weaver, Johanna Amann, Jethro Beekman, Mathias Payer, and Vern Paxson. The matter of heartbleed. In *2014 Internet Measurement Conference, IMC 2014, Vancouver, Canada, November 5-7, 2014*, pages 475–488, 2014.

[FBBP18] Benjamin Farinier, Sébastien Bardin, Richard Bonichon, and Marie-Laure Potet. Model generation for quantified formulas: A taint-based approach. In *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part II*, pages 294–313, 2018.

[FDBL18] Benjamin Farinier, Robin David, Sébastien Bardin, and Matthieu Lemerre. Arrays made simpler: An efficient, scalable and thorough preprocessing. In *LPAR-22. 22nd International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Awassa, Ethiopia, 16-21 November 2018*, pages 363–380, 2018.

[GKS05] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: directed automated random testing. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12-15, 2005*, pages 213–223, 2005.

[hea14] The heartbleed bug. http://heartbleed.com/, 2014.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[KKP+15] Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-c: A software analysis perspective. *Formal Asp. Comput.*, 27(3):573–609, 2015.

[KS08] Daniel Kroening and Ofer Strichman. *Decision Procedures - An Algorithmic Point of View*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2008.

[SMA05] Koushik Sen, Darko Marinov, and Gul Agha. CUTE: a concolic unit testing engine for C. In *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2005, Lisbon, Portugal, September 5-9, 2005*, pages 263–272, 2005.

[WMMR05] Nicky Williams, Bruno Marre, Patricia Mouy, and Muriel Roger. Pathcrawler: Automatic generation of path tests by combining static and dynamic analysis. In *Dependable Computing - EDCC-5, 5th European Dependable Computing Conference, Budapest, Hungary, April 20-22, 2005, Proceedings*, pages 281–292, 2005.

Chapter 2

Motivation

Chapter 1 gave an intuition of the false positives issue for Symbolic Execution, especially in the case of Symbolic Execution applied to vulnerability analysis. In this chapter we highlight this issue with a case study inspired by a real vulnerability discovered in 2015 in the GRUB2 bootloader. We then explain how to overcome this issue by taking into account the fact that some inputs are not controlled by the user, and by switching from a C-level semantics to a binary-level semantics, but at the price of generating larger constraints involving quantifiers. Because most solvers are not able to solve these generated large quantified constraints, we finally use the taint-based quantifier elimination presented in Chapter 5 and the simplification dedicated to array terms presented in Chapter 6 to make the constraints simple enough to be handled by usual solvers.

2.1 The Back to 28 Vulnerability

Let us consider as a motivating example the program given in Figure 2.1 and Figure 2.2, a simplified version of a security vulnerability named Back to 28 [MR15]. In its original version, it allowed to bypass the GRUB2 bootloader authentication procedure by pressing 28 times the backspace key. For the purpose of this example, we will merely corrupt in-memory data to bypass a password authentication. The program code is composed as follows:

```
char get_key (char buf[], int buf_size, int pos) {
  if (pos < 0 || pos >= buf_size) return 0;
  return buf[pos];
}

void sanitize (char buf[], int buf_size) {
  for (int i = 0; i < buf_size; i++) {
    buf[i] = 0;
  }
  return;
}

void read_input (char src[], int src_size, char dst[], int dst_size) {
  unsigned cur_len = 0;
  unsigned cur_pos = 0;
  char cur_key;

  while (cur_pos < src_size) {
    cur_key = get_key(src, src_size, cur_pos++);

    if (cur_key == 0 || cur_key == '\n' || cur_key == '\r') {
      break;
    }
    if (cur_key == '\b') {
      cur_len--;
      continue;
    }
    if (cur_len < dst_size) {
      dst[cur_len++] = cur_key;
    }
  }
  sanitize(dst + cur_len, dst_size - cur_len);
  return;
}
```

Figure 2.1

```
#define SIZE


void get_secret (char buf[], int buf_size) {
// Retrieve the secret
}

int check_password (char secr[], char pass[]) {
  int b = 1;
  for (int i = 0; i < SIZE; i++) {
    b &= secr[i] == pass[i];
  }
  return b;
}

int main (int argc, char *argv[]) {
  char inpt[2 * SIZE];
  char pass[SIZE];
  char secr[SIZE];

  if (argc != 2) return 0;

  fgets(inpt, 2 * SIZE, stdin);
  get_secret(secr, SIZE);
  read_input(inpt, 2 * SIZE, pass, SIZE);

  if (check_password(secr, pass)) {
    printf("Success!\n");
  }
  else {
    printf("Failure...\n");
  }
  return 0;
}
```

Figure 2.2

Figure 2.1 provides code for functions `get_key`, `sanitize` and `read_input`. The `read_input` function takes as parameter an input buffer and an output buffer, as well as their respective size. It reads in the first buffer characters entered by the user and writes in the second the result after having interpreted control characters: it stops reading the input if the read character is a line break, and it moves the write cursor back if the read character is a backspace. It calls the `get_key` function to make sure there is no read buffer overflow, and checks bounds before writing the output buffer in order to avoid write buffer overflows. Finally, it calls the `sanitize` function to clean the output buffer of characters coming after the write cursor.

Figure 2.2 provides code for functions `get_secret`, `check_password` and `main`. The body of the `get_secret` function is not given, but it is supposed that it writes, in the buffer passed as a parameter, a secret difficult to predict. The `check_password` function simply checks that the password provided by the user is equal to the secret. Finally, the `main` function retrieves the secret using the `get_secret` function and the user password using the `read_input` function. Then it checks if they match using the `check_password` function, and prints "Success!" if it is the case or prints "Failure..." if it is not.

The vulnerability lies in the fact that, although the bounds are checked before each write, the write cursor can still be moved out of the write buffer by repeatedly entering the backspace character. But the write cursor is transmitted to the `sanitize` function, which will clean the requested memory region by writing zeros, without checking bounds. We are in the presence of a write buffer overflow, but restricted to the writing of zeros.

Goal We aim to find an input for which the execution reaches the "Success!" branch. We show first how classic Symbolic Execution at C-level produces a non-reproducible solution, in other words produces a false positive. We also demonstrate how C-level Symbolic Execution can be made robust in order to avoid this false positives pitfall, but still without finding a "real" solution. We then consider a binary-level semantics of the program. If binary-level classic Symbolic Execution produces again a false positive, robust Symbolic Execution at binary-level reaches this time a "real" solution through the buffer overflow in the `sanitize` function.

2.2 First Attempt: High-Level Semantics

So let us try with C-level classic Symbolic Execution to find an input for which the execution reaches the "Success!" branch. Since the content of characters array `impt` comes from a user input, it is modeled by a first symbolic variable i. Similarly, the content of characters array `secr` is modeled by a second symbolic variable s, as it comes from the execution of a code which is not part of the analysis (call to an external cryptographic library, etc).

The content of characters array `pass` results from the execution of the `read_input` function. Therefore Symbolic Execution starts exploring this function, entering the `while` loop and then the body of the first `if` statement. To do so it has to satisfy the `if` statement condition `(cur_key == 0 || cur_key == '\n' || cur_key == '\r')` — the `while` condition is trivial here. `cur_key` is at that point in the execution equal to the first character of the input `inpt[0]`, whose symbolic counterpart is $i[0]$. Symbolic Execution thus generates the constraint $(i[0] = 0 \vee i[0] = \texttt{'\textbackslash n'} \vee i[0] = \texttt{'\textbackslash r'})$. Then Symbolic Execution reaches the `break` statement which breaks the `while` loop and makes the execution jump to the `sanitize` function. Because `cur_len` was left unchanged and is still equal to zero, the `sanitize` function writes zeros from `&pass` to `&pass` + `SIZE` − 1, i.e. fills array `pass` with zeros.

Once the `read_input` function executed, Symbolic Execution returns to the `main` function. In order to be able to reach the "Success!" branch, Symbolic Execution has to satisfy the `check_password` condition, which requires having `secr` equals to `pass`. But the latter was just filled with zeros. Therefore Symbolic Execution generates the constraint $s[0, \texttt{SIZE} - 1] = 0$, where $s[0, \texttt{SIZE} - 1]$ stands for values in s at indices ranging from 0 to `SIZE` − 1.

In the end, the full constraint that inputs have to satisfy is:

$$\begin{aligned}&\exists i.\exists s.\\ &\quad (i[0] = 0 \vee i[0] = \texttt{'\textbackslash n'} \vee i[0] = \texttt{'\textbackslash r'})\\ &\quad \wedge s[0, \texttt{SIZE} - 1] = 0\end{aligned}$$

But this constraint has a trivial solution $\{i[0, \texttt{SIZE} - 1] = 0, s[0, \texttt{SIZE} - 1] = 0\}$, which can be interpreted as "if the secret is empty, then let the input empty". Which would be correct, but will not happen in practice. *Thus it is not a real solution to our problem, it is a false positive.*

This false positive comes from the fact that Symbolic Execution considers all entries of the program in the same way, as if they were controlled by the user. This is reflected

in the path constraint by the two existential quantifications over i and s. But here, since the secret `secr` is not controlled by the user, an appropriate solution of the problem should be valid for all values of `secr`, i.e. for all values of s. Therefore *a better model of the problem would universally quantify over values of s*, which would result in the path constraint:

$$\begin{aligned}&\exists i.\forall s.\\&\quad (i[0] = 0 \vee i[0] = \texttt{'\textbackslash n'} \vee i[0] = \texttt{'\textbackslash r'})\\&\quad \wedge s[0, \texttt{SIZE} - 1] = 0\end{aligned}$$

This constraint has no solution anymore, but at least we are free of false positives.

2.3 Second Attempt: Low-Level Semantics

In this example the authentication bypass is achieved by a buffer overflow, which is an undefined behavior and has no semantics at C-level. Therefore if we want to find a solution to our problem, we must move to a binary-level semantics of our program and go into more details on the Symbolic Execution machinery:

First we notice that the `read_input` function reads the user input until it encounters an end-of-string character or reaches the end of the input buffer. On the Symbolic Execution side, this is materialized by the introduction of a constant N implicitly defined by the number of iterations spent in the reading loop and representing the length of the user input.

Second we notice that local variables `inpt`, `secr` and `pass` are fixed-size character arrays. According to our binary-level semantics, these arrays are directly allocated into the program stack, successively. On the Symbolic Execution side, this is materialized by the introduction of two fresh symbolic variables, m_0 representing the program memory at the beginning of its execution, and p_0 a pointer to the location in memory of the program stack. As for the secret, the initial program memory and the initial stack pointer are not controlled by the user, and thus m_0 and p_0 as s have to be universally quantified.

Once again Symbolic Execution starts exploring the `read_input` function by enumerating its feasible paths. But because of universal quantifications, most of these paths produce constraints with no solution and are quickly cut off. At some point, Symbolic Execution will analyze the path passing $N - 1$ times through the second `if` statement, before passing through the first one and leaving the `read_input` function. Denoting by

$\triangleq$ the definition of intermediate variables, Symbolic Execution will generate for this path the following constraint:

$$
\begin{aligned}
&\exists i.\, \forall s.\, \forall m_0.\, \forall p_0.\\
&\quad p_1 \triangleq p_0 - 2 \cdot \texttt{SIZE}\\
&\quad p_2 \triangleq p_1 - \texttt{SIZE}\\
&\quad p_3 \triangleq p_2 - \texttt{SIZE}\\
&\quad m_1 \triangleq m_0[p_1,\, p_1 + 2 \cdot \texttt{SIZE} - 1] \leftarrow i\\
&\quad m_2 \triangleq m_1[p_3,\, p_3 + \texttt{SIZE} - 1] \leftarrow s\\
&\quad m_3 \triangleq m_2[p_2 - (N-1)\,,\, p_2 + \texttt{SIZE} - 1] \leftarrow 0\\
&(i[0] = \texttt{'\textbackslash b'}) \wedge \cdots \wedge (i[N-2] = \texttt{'\textbackslash b'})\\
&\wedge (i[N-1] = 0 \vee i[N-1] = \texttt{'\textbackslash n'} \vee i[N-1] = \texttt{'\textbackslash r'})\\
&\wedge (m_3[p_2,\, p_2 + \texttt{SIZE} - 1] = m_3[p_3,\, p_3 + \texttt{SIZE} - 1])
\end{aligned}
$$

where:

- Variable i is existentially quantified while variables s, m_0 and p_0 are universally quantified because the user has control over the input but not over the secret, the initial memory nor the initial stack pointer: we look for a solution which is valid for all of their values.

- $p_1 \triangleq p_0 - 2 \cdot \texttt{SIZE}$ is a pointer to the `inpt` array, $p_2 \triangleq p_1 - \texttt{SIZE}$ is a pointer to the `secr` array, and $p_3 \triangleq p_2 - \texttt{SIZE}$ is a pointer to the `pass` array.

- $m_1 \triangleq m_0[p_1, p_1 + 2 \cdot \texttt{SIZE} - 1] \leftarrow i$ denotes the memory after writing the user input i into array `inpt`, $m_2 \triangleq m_1[p_3, p_3 + \texttt{SIZE} - 1] \leftarrow s$ denotes the memory after retrieving and writing the secret s into array `secr`, and $m_3 \triangleq m_2[p_2 - (N-1)\,,\, p_2 + \texttt{SIZE} - 1] \leftarrow 0$ denotes the memory after executing the `read_input` function. Indeed, because the second `if` statement was executed $N-1$ times, variable `cur_len` is equal to $1 - N$, and the `sanitize` function writes zeros from `&pass` + `cur_len` $\triangleq p_2 - (N-1)$ to `&pass` + `cur_len` + `SIZE` $- 1 -$ `cur_len` $\triangleq p_2 + \texttt{SIZE} - 1$.

- Finally, $(i[0] = \texttt{'\textbackslash b'}) \wedge \cdots \wedge (i[N-2] = \texttt{'\textbackslash b'})$ expresses the constraint for passing $N-1$ times through the second `if` statement, $i[N-1] = 0 \vee i[N-1] = \texttt{'\textbackslash n'} \vee i[N-1] = \texttt{'\textbackslash r'}$ expresses the constraint for passing one time through the first one, and $m_3[p_2, p_2 + \texttt{SIZE} - 1] = m_3[p_3, p_3 + \texttt{SIZE} - 1]$ expresses the constraint for having `secr` = `pass` and reaching the "Success!" branch.

However, this constraint involves universal quantifiers, which significantly increase the difficulty of finding a solution. In practice, most solvers fail to solve this constraint: we have to eliminate these universal quantifications. To that extend *we make use of the taint-based approach [FBBP18] presented in Chapter 5* which stipulates that, in our specific case, we can replace universal quantifications over s and m_0 by existential quantifications by adding the constraint $N - 1 \geq \texttt{SIZE}$:

$$
\begin{aligned}
&\exists i.\, \exists s.\, \exists m_0.\, \forall p_0.\\
&\quad p_1 \triangleq p_0 - 2 \cdot \texttt{SIZE}\\
&\quad p_2 \triangleq p_1 - \texttt{SIZE}\\
&\quad p_3 \triangleq p_2 - \texttt{SIZE}\\
&\quad m_1 \triangleq m_0[p_1,\ p_1 + 2 \cdot \texttt{SIZE} - 1] \leftarrow i\\
&\quad m_2 \triangleq m_1[p_3,\ p_3 + \texttt{SIZE} - 1] \leftarrow s\\
&\quad m_3 \triangleq m_2[p_2 - (N-1)\ ,\ p_2 + \texttt{SIZE} - 1] \leftarrow 0\\
&(i[0] = \texttt{'\textbackslash b'}) \wedge \cdots \wedge (i[N-2] = \texttt{'\textbackslash b'})\\
&\wedge (i[N-1] = 0 \vee i[N-1] = \texttt{'\textbackslash n'} \vee i[N-1] = \texttt{'\textbackslash r'})\\
&\wedge (m_3[p_2,\ p_2 + \texttt{SIZE} - 1] = m_3[p_3,\ p_3 + \texttt{SIZE} - 1])\\
&\wedge (N - 1 \geq \texttt{SIZE})
\end{aligned}
$$

The next step consists in simplifying this constraint. Indeed, for the sake of clarity we give a simplified version of the constraint generated by Symbolic Execution. Due to the binary-level semantics we had to use, the real constraint involves about ten thousands of terms, with thousands of array reads and hundreds of array writes. These array operations are known to be difficult to handle by solvers. Therefore *we apply a treatment dedicated to array terms [FDBL18] presented in Chapter 6* and obtain the following constraint, where only remains the existentially quantified variable i:

$$
\begin{aligned}
&\exists i.\\
&\quad (i[0] = \texttt{'\textbackslash b'}) \wedge \cdots \wedge (i[N-2] = \texttt{'\textbackslash b'})\\
&\quad \wedge (i[N-1] = 0 \vee i[N-1] = \texttt{'\textbackslash n'} \vee i[N-1] = \texttt{'\textbackslash r'})\\
&\quad \wedge (N - 1 \geq \texttt{SIZE})
\end{aligned}
$$

For example with `SIZE` = 8, {i[0, `SIZE`+1] = "`\b\b\b\b\b\b\b\b\n`"} is a possible solution, *which indeed leads to the "Success!" branch through the buffer overflow in* `sanitize` *function.* By pressing `SIZE` times the backspace key we have `cur_len` = −`SIZE`, which forces the `sanitize` function writting zeros from `&pass` − `SIZE` = `&secr` to `&pass` + `SIZE` − 1. Being both overwritten with zeros, array `pass` and array `secr` now contain the same values, making the `check_password` condition true.

Table 2.1 – Summary table of the characteristics of the various Symbolic Execution variations we presented in this chapter.

	QF constraints	has false positives	finds a solution	is handled by solvers
classic C-level SE	✓	✓	✓	✓
robust C-level SE	✗	✗	✗	✓
robust binary-level SE	✗	✗	✗	✗
robust binary-level SE + elim. + simpl.	✓	✗	✓	✓

2.4 Conclusion

In this chapter we brought to light the false positives problem which appears when applying Symbolic Execution to vulnerability analysis. We first show how on an authentication bypass scenario a C-level classic Symbolic Execution answers with an incorrect solution. The error stems from the fact that classic Symbolic Execution models every external component in the same way, with no distinction between those controlled by the user and those uncontrolled. So we made Symbolic Execution robust to false positives by distinguishing between the two, existentially quantifying variables which model controlled components and universally quantifying variables which model uncontrolled components. By doing so, C-level robust Symbolic Execution did not answer with an incorrect solution anymore, but still failed to find a correct solution.

In order to find a solution to our scenario, we then switched to a binary-level robust Symbolic Execution. Indeed, the authentication bypass is achieved by a buffer overflow, which is an undefined behavior with regard to the C-level semantics of the program, while being well-defined with regard to its binary-level semantics. But the problem now is that quantifications we introduced to distinguish controlled and uncontrolled variables, combined with the verbosity of the binary-level semantics of the program, make constraints generated by Symbolic Execution almost impossible solve.

For this reason, we used first a taint-based quantifier elimination [FBBP18] to remove universal quantifications over array variables, and second a simplification dedicated to array terms [FDBL18] to drastically reduce the size of the formula. By this means generated constraints became simple enough to be handled by usual SMT solvers. A summary of the characteristics of these Symbolic Execution variations is given in Table 2.1. Finally our robust Symbolic Execution was able to answer with a correct solution, which indeed led to an authentication bypass through a buffer overflow.

Bibliography

[FBBP18] Benjamin Farinier, Sébastien Bardin, Richard Bonichon, and Marie-Laure Potet. Model generation for quantified formulas: A taint-based approach. In *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part II*, pages 294–313, 2018.

[FDBL18] Benjamin Farinier, Robin David, Sébastien Bardin, and Matthieu Lemerre. Arrays made simpler: An efficient, scalable and thorough preprocessing. In *LPAR-22. 22nd International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Awassa, Ethiopia, 16-21 November 2018*, pages 363–380, 2018.

[MR15] Hector Marco and Ismael Ripoll. Back to 28: Grub2 authentication 0-day. http://hmarco.org/bugs/CVE-2015-8370-Grub2-authentication-bypass.html, 2015.

Part II

Background

Chapter 3

Many-Sorted First-Order Logic

The *many-sorted first-order logic* is a formal system which describes *many-sorted first-order formulas* and *many-sorted first-order theories* built upon. A formula is defined over *terms* of some *sorts*, which are the objects we want to prove a property on. For example in arithmetic, a term might represent an integer and therefore will belong to the integer sort. In linear algebra, a term might represent a vector space and therefore will belong to the vector space sort. A *formula* defines a property of the objects being discussed. For example in algebra we can express a formula which states the existence of the additive inverse. In arithmetic we can express a formula which states that the sum of two positive integers is positive. Finally a *theory* is a finite or an infinite set of formulas over the same sort of terms. For example Peano axioms define a theory of arithmetic, and Zermelo-Fraenkel axioms define a set theory commonly used in mathematics.

Because most automatic software analyzers use it to formally reason about programs, we give in this chapter a thorough description of the many-sorted first-order logic. In particular, Symbolic Execution we present in Chapter 4 translates program execution traces into formulas interpreted over the bit-vectors and arrays theory. We first define the syntax and semantics of the many-sorted first-order logic Section 3.1. Then we present a general decision procedure for many-sorted first-order formulas Section 3.2. Finally we introduce three many-sorted first-order theories pervasive in software analysis Section 3.3.

3.1 Syntax and Semantics

Formulas are symbol sequences following precise syntactic rules and defining a language. Some symbols are common to all languages, like logical constants, connectors

and quantifiers. Others are characteristics and form the signature of a language. For example in group theory we need a constant symbol to represent the neutral element and a binary symbol to represent the group law. Formulas we consider are first-order, which means that quantification is limited to elements belonging to some sort. This restriction forbids us for example to quantify over sets of elements or over functions over elements. The many-sorted aspect of our formulas mitigates this constraint. It means that formulas may contain terms of different sorts, and therefore allows to have a sort for sets of elements of some fixed sorts, or for functions over elements of some fixed sorts.

In Section 3.1.1 we define many-sorted first-order formulas built from variables, function symbols, predicate symbols, logical connectives and quantifiers. In Section 3.1.2 we define interpretations and models, and what it means to be valid or satisfiable. Finally, in Section 3.1.3 we define what are theories and refine, modulo theory, the meaning of validity and satisfiability.

3.1.1 Signatures, Terms and Formulas

The following first definition simply states that a language is characterized by the list of symbols it uses.

Definition 3.1 (Signature)**.** A (many-sorted first-order) *signature* is a triple $\Sigma = (\mathcal{S}_\Sigma, \mathcal{F}_\Sigma, \mathcal{P}_\Sigma)$ where:

- $\mathcal{S}_\Sigma$ is a set of *sort symbols*,
- $\mathcal{F}_\Sigma$ is a set of *function symbols* $f : s_1 \times \cdots \times s_n \to s$, with n the arity of f, with $s_1 \times \cdots \times s_n \in \mathcal{S}_\Sigma^n$ the expected parameters sorts of f, and with $s \in \mathcal{S}_\Sigma$ the result sort of f,
- $\mathcal{P}_\Sigma$ is a set of *predicate symbols* $p : s_1 \times \cdots \times s_n$, with n the arity of p, and with $s_1 \times \cdots \times s_n \in \mathcal{S}_\Sigma^n$ the expected parameters sorts of p.

A symbol with a zero arity is called a *constant symbol*. Note that we intentionally left missing the result sort for predicate symbols, as they all belong to the world of predicates.

Terms are objects language is talking about.

Definition 3.2 (Term)**.** Let $\Sigma = (\mathcal{S}_\Sigma, \mathcal{F}_\Sigma, \mathcal{P}_\Sigma)$ be a signature, and let $(\mathcal{V}_s)_{s \in \mathcal{S}_\Sigma}$ be a family of countably infinite set of sorted variables. A (many-sorted first-order) Σ-term of sort $s \in \mathcal{S}_\Sigma$ is recursively defined as a variable, or a function symbol application:

$$
\begin{array}{lll}
t & = & \\
& \mid & x \qquad\qquad\quad \text{with } x \in \mathcal{V}_s \\
& \mid & f\,(t_1, \ldots, t_n) \quad \text{with } f : s_1 \times \cdots \times s_n \to s \in \mathcal{F}_\Sigma \\
& & \qquad\qquad\quad \text{and for all } i \in [1, n],\ t_i \text{ a term of sort } s_i
\end{array}
$$

We build formulas from atomic formulas using logical constants, connectors and quantifiers. These usually include constants $\top$ (true) and $\bot$ (false), connectors $\neg$ (negation), $\vee$ (disjunction), $\wedge$ (conjunction) and $\Rightarrow$ (implication), and quantifiers $\exists$ (existential) and $\forall$ (universal). But as they can all be defined from $\bot$, $\Rightarrow$ and $\forall$, we only consider the latter in following definitions.

Definition 3.3 (Formula)**.** Let $\Sigma = (\mathcal{S}_\Sigma, \mathcal{F}_\Sigma, \mathcal{P}_\Sigma)$ be a signature, and let $(\mathcal{V}_s)_{s \in \mathcal{S}_\Sigma}$ be a family of countably infinite set of sorted variables. A (many-sorted first-order) Σ-formula is recursively defined as the false formula, a universal quantification, a logical implication, or a predicate symbol application:

$$
\begin{array}{lll}
f & = & \\
& \mid & \bot \\
& \mid & \forall x : s.f \qquad\quad \text{with } s \in \mathcal{S}_\Sigma,\ x \in \mathcal{V}_s \text{ and } f \text{ a formula} \\
& \mid & f_1 \Rightarrow f_2 \qquad \text{with } f_1 \text{ and } f_2 \text{ two formulas} \\
& \mid & p\,(t_1, \ldots, t_n) \quad \text{with } p : s_1 \times \cdots \times s_n \in \mathcal{P}_\Sigma \\
& & \qquad\qquad\quad \text{and for all } i \in [1, n],\ t_i \text{ a term of sort } s_i
\end{array}
$$

A formula not containing quantifiers is said to be *quantifier-free*. Logical constants and predicate symbol applications are called *atoms* or atomic formulas.

Example. Let us consider the Integer Difference Logic (IDL), defined over the signature $(\mathcal{S}_{\text{IDL}}, \mathcal{F}_{\text{IDL}}, \mathcal{P}_{\text{IDL}})$ with $\mathcal{S}_{\text{IDL}} = \{\text{Int}\}$, $\mathcal{F}_{\text{IDL}} = \{-, 0, 1, \ldots\}$, and $\mathcal{P}_{\text{IDL}} = \{>\}$. The set of sort symbols $\mathcal{S}_{\text{IDL}}$ contains a single sort symbol, the integer sort symbol Int. The set of function symbols $\mathcal{F}_{\text{IDL}}$ contains a binary function symbol $-$ representing the subtraction, and an infinity of constant function symbols $0, 1, \ldots$ representing constant integer values. Finally the set of predicate symbols $\mathcal{P}_{\text{IDL}}$ contains a single binary predicate symbol, the comparison symbol $>$.

The formula $\forall x : \text{Int}.\ \forall y : \text{Int}.\ (x - y > 0) \Leftrightarrow (x > y)$ contains four terms: the subtraction $x - y$, the constant 0, and two universally quantified integer variables x and y. Two atoms are built upon these terms: $(x - y > 0)$ and $(x > y)$, linked together by an equivalence relation $\Leftrightarrow$. Note that $\phi \Leftrightarrow \psi$ is a shortand for $(\phi \Rightarrow \psi) \wedge (\psi \Rightarrow \phi)$.

Definition 3.4 (Bound and Free Variable)**.** In the formula $\forall x : s.f$, f is called the *scope* of the quantification $\forall x : s$. An occurrence of a variable is said to be *bound* if it lies within the scope of a quantifier. Otherwise, the occurrence is said to be *free*. Formulas without free variables are called *closed formulas*.

Quantifiers bring some problems about variable names. For example, formulas $\forall x : s.\forall y : s.xy = yx$ and $\forall y : s.\forall x : s.yx = xy$ are equal up to renaming. They are said to be α-equivalent. But they are not α-equivalent to $\forall x : s.\forall x : s.xx = xx$, where the second $\forall x : s$ shadows the first one. In the following we always work up to α-equivalence and consider that variable names are non-conflicting.

3.1.2 Interpretations and Models

Now that we have defined the syntax of formulas, we have to give them a semantics and to define what it means for a formula to be true. To this end, we have to interpret every symbol in the formula. An *interpretation* for a sort symbol is a set of concrete elements called domain, and an interpretation for a function or a predicate symbol will be a concrete function or predicate.

Definition 3.5 (Interpretation)**.** Let $\Sigma = (\mathcal{S}_\Sigma, \mathcal{F}_\Sigma, \mathcal{P}_\Sigma)$ be a signature. A Σ-interpretation is a triple $\mathcal{I} = (\llbracket \mathcal{S}_\Sigma \rrbracket_\mathcal{I}, \llbracket \mathcal{F}_\Sigma \rrbracket_\mathcal{I}, \llbracket \mathcal{P}_\Sigma \rrbracket_\mathcal{I})$ with:

- $\llbracket \mathcal{S}_\Sigma \rrbracket_\mathcal{I} = \{\llbracket s \rrbracket_\mathcal{I} \mid s \in \mathcal{S}_\Sigma\}$, where each $\llbracket s \rrbracket_\mathcal{I}$ is a set interpreting its associated sort symbol s,
- $\llbracket \mathcal{F}_\Sigma \rrbracket_\mathcal{I} = \left\{\llbracket f \rrbracket_\mathcal{I} : \llbracket s_1 \rrbracket_\mathcal{I} \times \cdots \times \llbracket s_n \rrbracket_\mathcal{I} \to \llbracket s \rrbracket_\mathcal{I} \mid f : s_1 \times \cdots \times s_n \to s \in \mathcal{F}_\Sigma\right\}$, where each $\llbracket f \rrbracket_\mathcal{I}$ is a function interpreting its associated function symbol f,
- $\llbracket \mathcal{P}_\Sigma \rrbracket_\mathcal{I} = \left\{\llbracket p \rrbracket_\mathcal{I} : \llbracket s_1 \rrbracket_\mathcal{I} \times \cdots \times \llbracket s_n \rrbracket_\mathcal{I} \mid p : s_1 \times \cdots \times s_n \in \mathcal{P}_\Sigma\right\}$, where each $\llbracket p \rrbracket_\mathcal{I}$ is a predicate interpreting its associated predicate symbol p.

In order to determine the truth value of a formula involving variables, we have to state which objects variables refer to. This relates to the notion of assignment.

Definition 3.6 (Assignment)**.** Let $\Sigma = (\mathcal{S}_\Sigma, \mathcal{F}_\Sigma, \mathcal{P}_\Sigma)$ be a signature, let $\mathcal{I}$ be a Σ-interpretation, and let $(\mathcal{V}_s)_{s \in \mathcal{S}_\Sigma}$ be a family of countably infinite set of sorted variables. An assignment is a map family $(\alpha_s : \mathcal{V}_s \to \llbracket s \rrbracket_\mathcal{I})_{s \in \mathcal{S}_\Sigma}$ from sorted variables to elements of interpretation domains.

Let α be an assignment, $x \in \mathcal{V}_s$ and $a \in \llbracket s \rrbracket_\mathcal{I}$ for some s. Then $\alpha\,[x \mapsto a]$ denotes the assignment:

$$\alpha\,[x \mapsto a]\,(y) = \begin{cases} a & \text{if } x = y \\ \alpha\,(y) & \text{otherwise} \end{cases}$$

Given a signature, an interpretation and an assignment, we can now define the value of a term and the truth value of a formula.

Definition 3.7 (Term Value)**.** Let Σ be a signature, $\mathcal{I}$ be a Σ-interpretation, and α be an assignment. The value of a Σ-term is recursively defined as:

$$\begin{aligned} \mathrm{Val}_{\mathcal{I}}(x, \alpha) &= \alpha(x) \\ \mathrm{Val}_{\mathcal{I}}(f(t_1, \ldots, t_n), \alpha) &= [\![f]\!]_{\mathcal{I}}(\mathrm{Val}_{\mathcal{I}}(t_1, \alpha), \ldots, \mathrm{Val}_{\mathcal{I}}(t_n, \alpha)) \end{aligned}$$

Definition 3.8 (Formula Truth Value)**.** Let Σ be a signature, $\mathcal{I}$ be a Σ-interpretation, and α be an assignment. The truth value of a Σ-formula is recursively defined as:

$$\begin{aligned} \mathrm{Val}_{\mathcal{I}}(\bot, \alpha) &= \bot \\ \mathrm{Val}_{\mathcal{I}}(\forall x : s.f, \alpha) &= \forall a \in [\![s]\!]_{\mathcal{I}}.\mathrm{Val}_{\mathcal{I}}(f, \alpha[x \mapsto a]) \\ \mathrm{Val}_{\mathcal{I}}(f_1 \Rightarrow f_2, \alpha) &= \mathrm{Val}_{\mathcal{I}}(f_1, \alpha) \Rightarrow \mathrm{Val}_{\mathcal{I}}(f_2, \alpha) \\ \mathrm{Val}_{\mathcal{I}}(p(t_1, \ldots, t_n), \alpha) &= [\![p]\!]_{\mathcal{I}}(\mathrm{Val}_{\mathcal{I}}(t_1, \alpha), \ldots, \mathrm{Val}_{\mathcal{I}}(t_n, \alpha)) \end{aligned}$$

Note that for a closed formula the choice of the initial assignment does not matter as quantifiers will override it systematically. Subsequently, we can always choose the empty assignment $\varnothing$ to evaluate the truth value of a closed formula.

We end up with the definition of models, which are interpretations making a formula true. A formula is said to be satisfiable if it admits a model, i.e. if there exists an interpretation which makes the formula true. And a formula is said to be valid if any interpretation is a model, i.e. the formula is true for all interpretations.

Definition 3.9 (Model, Satisfiability and Validity)**.** Let Σ be a signature, and f be a Σ-formula. A Σ-interpretation $\mathcal{I}$ is a *model* of f if there exists an assignment α such that $\mathrm{Val}_{\mathcal{I}}(f, \alpha)$ is true, denoted $\mathcal{I}, \alpha \models f$.

A Σ-formula f is said to be:

- *satisfiable* if there exists $\mathcal{I}$ and α such that $\mathcal{I}, \alpha \models f$;
- *unsatisfiable* if for all $\mathcal{I}$ and α, $\mathcal{I}, \alpha \not\models f$;
- *valid* if for all $\mathcal{I}$ and α, $\mathcal{I}, \alpha \models f$;
- *invalid* if there exists $\mathcal{I}$ and α such that $\mathcal{I}, \alpha \not\models f$.

Because the truth value of a closed formula does not depend on chosen assignment, we can omit it and simply denote a model of a closed formula by $\mathcal{I} \models f$.

Note that satisfiability and validity are dual notions. A formula f is satisfiable if and only if $\neg f$ is invalid, and f is valid if and only if $\neg f$ is unsatisfiable. For this reason, it is common to reduce the question of truth for a formula to the question of its (un)satisfiability.

3.1.3 Satisfiability Modulo Theories

As previously stated, a formula is satisfiable as soon as it admits a model. However in practice, we do not want to consider arbitrary models, but rather prefer to consider those where symbol interpretations respect the axioms of the theory we study. A many-sorted first-order theory is a set of many-sorted first-order formulas — the axioms of the theories — whose purpose is to restrict eligible models for formulas of this theory. A formula is satisfiable with respect to a theory if it admits a model which also satisfies all the axioms of the theory. Thus we speak of satisfiability modulo theory.

Definition 3.10 (Theory)**.** Let Σ be a signature. A Σ-*theory* $\mathcal{T}$ is a countable set of closed Σ-formula. A Σ-interpretation is a $\mathcal{T}$-model if it is a model of all formulas in $\mathcal{T}$. $\mathcal{T}$ is said to be *consistent* (or *non-contradictory*) if it admits a model, *inconsistent* (or *contradictory*) if it admits no model. A Σ-interpretation is a $\mathcal{T}$-model of f if it is a $\mathcal{T}$-model and a model of f, denoted $\mathcal{I} \models_{\mathcal{T}} f$.

A Σ-formula f is:

- *satisfiable* in $\mathcal{T}$ if there exists $\mathcal{I}$ such that $\mathcal{I} \models_{\mathcal{T}} f$;
- *unsatisfiable* in $\mathcal{T}$ if for all $\mathcal{I}$, $\mathcal{I} \not\models_{\mathcal{T}} f$;
- *valid* in $\mathcal{T}$ if for all $\mathcal{I}$, $\mathcal{I} \models_{\mathcal{T}} f$;
- *invalid* in $\mathcal{T}$ if there exists $\mathcal{I}$ such that $\mathcal{I} \not\models_{\mathcal{T}} f$.

Example. The formula $\forall x : \text{Int}.\ \forall y : \text{Int}.\ (x - y > 0) \Leftrightarrow (x > y)$ is satisfiable but not valid. Indeed, the formula is true if we give to each symbol its usual interpretation, but is false if, for example, we interpret the subtraction symbol $-$ as the multiplication. However the formula is IDL-valid as axioms of the Integer Difference Logic force to interpret symbols with their usual meaning.

3.2 Deciding Many-Sorted First-Order Logic

We now look into the satisfiability modulo theories problem: *given a formula in some theory, does this formula admit a model or not?* The difficulty of this decision problem is tightly linked to the expressiveness of the chosen theory. In fact, the satisfiability problem for first-order logic (without theories) is undecidable, and this undecidability spreads to most of quantified theories — but not all of them. For example, the theory of Presburger arithmetic defined by the signature $\{0, 1, +, =\}$ is decidable, while the theory of Peano arithmetic defined by the signature $\{0, 1, +, \times, =\}$ is undecidable. In

this case we can try to restrict ourselves to the quantifier-free fragment of the theory, which is often easier to decide.

In this section we depict a general decision procedure for the satisfiability modulo theories problem which is the basis of most modern SMT solvers. In Section 3.2.1 we introduce some normal forms that we will use thereafter. In Section 3.2.2 we present a general decision procedure for propositional logic. In Section 3.2.3 we explain how this procedure can be extend to decidable quantifier-free theories, and finally extend to quantified theories in Section 3.2.4.

3.2.1 Normal Forms

In this section we introduce normal forms that refer to certain syntactic properties of the formula. Most of the decision procedures expect a formula in a specific normal form. Therefore it is common to begin the process by turning the formula into the form that procedure is designed to work with. This transformation has to preserve satisfiability in order to keep the overall procedure correct.

Let us first define literals, a syntactic unit frequently used in definitions of normal forms.

Definition 3.11 (Literal)**.** Let $\Sigma = (\mathcal{S}_\Sigma, \mathcal{F}_\Sigma, \mathcal{P}_\Sigma)$ be a signature. A Σ-literal is either a predicate symbol application or its negation:

$$
\begin{array}{rcll}
l & = & & \\
& | & p\left(t_1, \ldots, t_n\right) & \text{with } p : s_1 \times \cdots \times s_n \in \mathcal{P}_\Sigma \\
& | & \neg p\left(t_1, \ldots, t_n\right) & \text{and for all } i \in [1, n],\ t_i \text{ a term of sort } s_i
\end{array}
$$

We now give four of the most recurrent normal forms: prenex, negative, disjunctive and conjunctive normal form.

Definition 3.12 (Prenex Normal Form)**.** A formula is said to be in *prenex normal form* (PNF) if it is in the form $Q_1 x_0 : s_0 \ldots Q_n x_n : s_n . f$, with for all $i \in [1, n]$ $Q_i \in \{\exists, \forall\}$ and f a quantifier-free formula.

Definition 3.13 (Negative Normal Form)**.** A quantifier-free formula is in *negative normal form* (NNF) if negation $\neg$ occurs only in literals.

A quantified formula is in negative normal form if it is in prenex normal form and its quantifier-free part is in negative normal form.

Definition 3.14 (Disjunctive Normal Form)**.** A quantifier-free formula is in *disjunctive normal form* (DNF) if it is a disjunction of conjunctions of literals, i.e. is in the form $\bigvee_i \left(\bigwedge_j l_{i,j}\right)$ with $l_{i,j}$ the jth literal in the ith conjunction.

A quantified formula is in disjunctive normal form if it is in prenex normal form and if its quantifier-free part is in disjunctive normal form.

Definition 3.15 (Conjunctive Normal Form)**.** A quantifier-free formula is in *conjunctive normal form* (CNF) if it is a conjunction of disjunctions of literals, i.e. is in the form $\bigwedge_i \left(\bigvee_j l_{i,j}\right)$ with $l_{i,j}$ the jth literal in the ith disjunction. A literals disjunction in a CNF formula is called a *clause*.

A quantified formula is in conjunctive normal form if it is in prenex normal form and if its quantifier-free part is in conjunctive normal form.

As already stated, turning a formula into the normal form that the decision procedure is designed to work with has to preserve satisfiability in order to keep the overall procedure correct. Fortunately, for all formulas ϕ, for any of the four normal forms just defined, there exists a formula ψ in that form such that ϕ and ψ are equisatisfiable in the following meaning.

Definition 3.16 (Equisatisfiability, Equivalence)**.** Let Φ and Ψ two formulas.

- Φ and Ψ are *equisatisfiable* if they are both satisfiable or they are both unsatisfiable: there exists $\mathcal{M} \models \Phi$ if and only if there exists $\mathcal{N} \models \Psi$.
- Φ and Ψ are *equivalent* if they always have the same models: for all $\mathcal{M}$, $\mathcal{M} \models \Phi$ if and only if $\mathcal{M} \models \Psi$.

If there always exists an equisatisfiable formula in the desired normal form, this formula might be exponentially larger than the original formula. Luckily there exist several tactics to avoid this size explosion. For the conjunctive normal form, the most famous of such tactics is the Tseitin's encoding. It allows to turn any formula into an equisatisfiable CNF formula of linear size, only at the price of introducing a linear number of fresh symbols.

Finally let us define another normal form commonly encountered in general quantified formulas resolution.

Definition 3.17 (Skolem Normal Form)**.** A quantified formula is said to be in *Skolem Normal Form* (SNF) if it is in prenex normal form and contains only universal quantifiers.

Every formula can be converted into an equisatisfiable formula in Skolem normal form by repeatedly applying the following procedure called Skolemization. Let $\forall x_1 \ldots \forall x_n . \exists y . \varphi\,(x_1, \ldots, x_n, y)$ be a formula, and f_y a fresh function symbol. Then remove the existential quantification $\exists y$ and replace in φ all occurrences of y by $f_y\,(x_1, \ldots, x_n)$. In other words, Skolemization turns existentially quantified variables into function symbols parameterized by universally quantified variables.

In the rest of this chapter we assume that formulas are turned into conjunctive normal form, using if needed Tseitin's encoding to avoid exponential size explosion.

3.2.2 Propositional Logic

We start by presenting a brief summary of the Conflict-Driven Clause Learning (CDCL) algorithm [SS96, JS97, SS99], designed to solve the satisfiability problem for propositional formulas, also known as the SAT problem. Propositional formulas are special cases of first-order formulas, involving no term and only constant predicates, called here propositional constants. As there are no terms, there are no variables, and therefore quantifiers are useless.

Definition 3.18 (Propositional Logic)**.** A formula in propositional logic is a quantifier-free Σ-formula where $\Sigma = (\mathcal{S}_\Sigma, \mathcal{F}_\Sigma, \mathcal{P}_\Sigma)$ with $\mathcal{S}_\Sigma = \varnothing$, $\mathcal{F}_\Sigma = \varnothing$ and for all $p \in \mathcal{P}_\Sigma$, p is a constant predicate symbol.

CDCL was developed over time as a series of improvements to the Davis-Putnam-Loveland-Logemann (DPLL) procedure [DP60, DLL62], presented in Algorithm 3.1 in its simplest form. Basically, it consists in a recursive enumeration of possible interpretations, making at each step a decision about a variable and its value. If at some point an interpretation satisfying all the clauses in the formula is found, then it is a model and the algorithm returns it. But if an interpretation invalidating any of the clauses in the formula is reached, then the algorithm backtracks to the previous decision point.

Taking as the basis the DPLL procedure, CDCL incorporates the following extensions.

Boolean Constraint Propagation When all literals but one in a still unsatisfied clause are assigned, the remaining literal has to be assigned so that the clause is satisfied. Such assignment may lead to other constrained assignments, enriching the interpretation and potentially yielding to early conflict detection. This is called Boolean Constraint Propagation, also known as Unit Propagation, and is performed each time a decision for an assignment is made.

Algorithm 3.1: The Davis-Putnam-Logemann-Loveland procedure.

```
Function DPLL (Φ, M):
    Input: Φ a CNF formula in propositional logic
    Input: M the model under construction, initially empty
    Output: SAT (M) with M ⊨ Φ or UNSAT
    if for all clause φ in Φ, M ⊨ φ then
        return SAT (M)
    else if for some clause φ in Φ, M ⊭ φ then
        return UNSAT
    else
        Let l some unassigned literal in Φ
        match DPLL (Φ, M ∪ {l = ⊤})
            with SAT (M) return SAT (M)
            with UNSAT
                return DPLL (Φ, M ∪ {l = ⊥})
```

Non-Chronological Backtrack When an assignment leads to a conflict, the DPLL procedure backtracks to the previous decision point. A better strategy is to backtrack to the second most recent decision in the invalidated clause, while erasing all decisions made after, and immediately performs the assignment opposite to the most recent decision. This is referred as Non-Chronological Backtrack.

Conflict Clause Learning Let us assume a decision implies through constraint propagation the variable x to be assigned to $\bot$. Let us assume further that a later dicision assigning y to $\top$ leads to a conflict. Then we can safely add to the formula the clause $(x \vee \neg y)$. As this clause is logically implied by the original formula, this addition does not change the status of the formula. But assigning x to $\bot$ immediately leads through constraint propagation to the variable y being assigned to $\bot$. This process is known as conflict clause learning and plays an essential role in modern solvers.

Literal Assignement Heuristic Probably the most important element is the decision heuristic use to choose variables and their values. We do not intend to survey all of known decisision strategies, as new heurisitics are published every year. But common strategies consider the frequency of appearance of variables, the size of

the clause in which they appear, or how recently a variable was part of a conflict.

3.2.3 Quantifier-Free Formulas Modulo Theories

We now extend the CDCL algorithm from propositional formulas to quantifier-free formulas modulo theories. This extension is known under the name of DPLL$_\mathcal{T}$ [Tin02, GHN+04], sometimes CDCL$_\mathcal{T}$, where $\mathcal{T}$ refers to the theory of formulas we aim to solve. The main interest of DPLL$_\mathcal{T}$ lay in splitting the decision problem into two parts: solving propositional formulas on one side, and solving conjunction of atomic $\mathcal{T}$-formulas on the other side. More effective versions of DPLL$_\mathcal{T}$ tightly intertwine theory related mechanisms into propositional ones, but the general idea is the same.

Algorithm 3.2: A naive DPLL$_\mathcal{T}$.

Function `Solver` (Φ)**:**
- **Input:** Φ a CNF formula in propositional logic
- **Output:** SAT $(\mathcal{M})$ with $\mathcal{M} \models \Phi$ or UNSAT

Function `Theory` (Φ)**:**
- **Input:** Φ a conjunction of atomic formulas in QF-$\mathcal{T}$
- **Output:** SAT $(\mathcal{M}_\mathcal{T})$ or UNSAT (Ψ) with $\models_\mathcal{T} \Psi$ contradicting Φ

Function `DPLL`$_\mathcal{T}$ (Φ)**:**
- **Input:** Φ a CNF formula in QF-$\mathcal{T}$
- **Output:** SAT $(\mathcal{M}_\mathcal{T})$ with $\mathcal{M}_\mathcal{T} \models_\mathcal{T} \Phi$ or UNSAT
- **match** `Solver` $(\rho\,(\Phi))$
  - **with** *UNSAT* **return** *UNSAT*
  - **with** *SAT* $(\mathcal{M})$
    - **match** `Theory` $(\pi\,(\Phi, \mathcal{M}))$
      - **with** *SAT* $(\mathcal{M}_\mathcal{T})$ **return** *SAT* $(\mathcal{M}_\mathcal{T})$
      - **with** *UNSAT* (Ψ)
        - **return** `DPLL`$_\mathcal{T}$ $(\Phi \wedge \Psi)$

In this section we assume we are able to solve conjunctions of atomic $\mathcal{T}$-formulas. Some decision procedures for such formulas will be presented in Section 3.3. The simplified DPLL$_\mathcal{T}$ procedure we present in Algorithm 3.2 involves four auxiliary functions: Solver, a propositional SAT solver; Theory, a solver dedicated to conjunctions of atomic $\mathcal{T}$-formulas; ρ, an encoder which associates to a $\mathcal{T}$-formula a propositional

formula; and π, an encoder which turns a model into a conjunction of atomic $\mathcal{T}$-formulas.

As an example, let us solve using the DPLL$_\mathcal{T}$ procedure the linear arithmetic formula $((x < 0) \vee (x < y)) \wedge ((x \geq 0) \vee (x \geq y)) \wedge (y > 0)$. The encoder ρ normalizes it into $((x < 0) \vee (x < y)) \wedge (\neg (x < 0) \vee \neg (x < y)) \wedge \neg (y < 0) \wedge \neg (y = 0)$ and returns the propositional formula $(A \vee B) \wedge (\neg A \vee \neg B) \wedge \neg C \wedge \neg D$, where A stands for $(x < 0)$, B for $(x < y)$, C for $(y < 0)$, and D for $(y = 0)$. This formula is send to a propositional SAT solver which returns a first model, $\{A = \top, B = \bot, C = \bot, D = \bot\}$ for example. From this model, the π encoder produces the conjunction of atomic formulas $(x < 0) \wedge \neg (x < y) \wedge \neg (y < 0) \wedge \neg (y = 0)$, which unfortunately is $\mathcal{T}$-unsatisfiable. Consequently its negation $\neg (x < 0) \vee (x < y) \vee (y < 0) \vee (y = 0)$ is a $\mathcal{T}$-valid clause and can be added to the original formula with no effect but forbidding the previous propositional model. The propositional solver is called again on the ρ-encoding of this new formula, returns this time a second model $\{A = \bot, B = \top, C = \bot, D = \bot\}$, which leads to the π-encoded formula $\neg (x < 0) \wedge (x < y) \wedge \neg (y < 0) \wedge \neg (y = 0)$. This final formula is satisfiable with model $\{x = 0, y = 1\}$, which indeed is a model of the original formula.

3.2.4 Quantified Formulas Modulo Theories

The last stage in our attempt to decide many-sorted first-order formulas consists in extending the decision procedure for quantifier-free formulas we just defined in Section 3.2.3 to quantified formulas. However the many-sorted first-order logic is undecidable in general, and undecidability implies that there is no general algorithm that solves all cases. Still, the problem is decidable for some classes of formulas, for example when the domain of quantified variables is finite, or if there exist some quantifier elimination algorithms, whereas general but incomplete algorithms which solve many useful cases can be designed as a fallback.

In this section we consider only theories whose quantifier-free part is decidable.

Enumeration The simplest decidable class of quantified formulas is the class of formulas where the domain of quantified variables is finite. In this case, we instantiate the quantified variables with each element of the domain. Then, we replace existential quantification by a disjunction of all of these instantiations, and universal quantification by a conjunction of all of these instantiations.

For example, let us consider the quantified boolean formula $\forall x : \text{Bool}.\exists y : \text{Bool}. (x \vee \neg y) \wedge (\neg x \vee y)$, where booleans are implicitly lifted into propositional constants. This formula becomes $\forall x : \text{Bool}. ((x \vee \neg\top) \wedge (\neg x \vee \top)) \vee ((x \vee \neg\bot) \wedge (\neg x \vee \bot))$ after

the expansion of $\exists y : \text{Bool}$ and is simplified into $\forall x : \text{Bool}.x \vee \neg x$. Then the expansion of $\forall x : \text{Bool}$ results in $(\top \vee \neg\top) \wedge (\bot \vee \neg\bot)$ simplified into $\top$ which is trivially valid.

Note that enumerating over the domain of quantified variables, although always theoretically possible for finite domains, might be impossible in practice for large domains, as it is the case with bit-vectors for example.

Quantifier Elimination A quantifier elimination algorithm transforms a quantified formula into an equivalent formula without quantifier. The enumeration mechanism we presented for quantification over finite domains can be seen as such algorithm. There also exist quantifier elimination algorithms for formulas with quantification over infinite (or finite but too large) domains. For example, the Fourier-Motzkin projection and its extensions [Bjø10] eliminate quantifications for disjunctive linear arithmetic formulas.

Of course, not every theory admits a quantifier elimination algorithm, as its existence implies the decidability of the theory.

Instantiation When enumeration and quantifier elimination are both impossible, we have to fallback to some more general but incomplete algorithms. Many of these algorithms rely on the concept of *instantiation*.

Let us consider CNF-formulas turned into Skolem Normal Form. As universal quantifications and conjunctions commute, the resulting universally quantified conjunction can be split in two parts: a set of universally quantified clauses and a set of ground clauses. First note that if the conjunction of ground clauses is unsatisfiable, then the formula as a whole is unsatisfiable. Instantiation techniques take their name from the fact that for any term t, $\forall x.\varphi\,(x) \models \varphi\,(t)$. Hence, we can instantiate any of the universally quantified clauses with ground terms and add the resulting ground clause without changing the satisfiability of the formula. And if, after several additions, the set of ground terms becomes unsatisfiable, then the original formula is proved to be unsatisfiable. However, it will never be possible to conclude to the satisfiability of the formula.

The question is now which term to choose to instantiate universally quantified terms. A simple strategy consists in identifying for each quantified term a set of triggers as sub-terms containing all quantified variables. Then ground terms are explored in order to find sub-terms that match a trigger. The corresponding quantified term is instantiated so that it unifies the trigger and the matched sub-term. More advanced instantiation strategies like E-matching [DNS05] do not only consider sub-terms when matching a ground term, but also terms known to be equivalent to those sub-terms.

3.3 Some Many-Sorted First-Order Theories

We assumed in Section 3.2.3 we were able to solve conjunctions of atomic quantifier-free $\mathcal{T}$-formulas. To close the loop, we present in this section three theories pervasive in software analysis, together with their syntax, their semantics and with dedicated decision procedures for their quantifier-free fragment: the equality logic with uninterpreted functions theory in Section 3.3.1, the bit-vectors theory in Section 3.3.2, and the arrays theory in Section 3.3.3. Finally in Section 3.3.4, given some disjoined theories coming with dedicated decision procedures, we explain how to decide the combination of these theories.

3.3.1 Equality Logic with Uninterpreted Functions

In this section we introduce the theory of equality, also known as equality logic, together with uninterpreted functions. Equality logic can be thought of as propositional logic extended with equalities between terms of the same sort. Because they make it far more useful, equality logic is usually combined with uninterpreted functions which are, as their name suggests, function symbols which should not be interpreted as part of a model of a formula. Uninterpreted functions are widely used for simplifying or generalizing formulas. They let us reason about the formula while ignoring the semantics of some functions. The only thing we need to satisfy is functional consistency, which states that, given the same inputs, a function returns the same outputs. At the same time, replacing functions with uninterpreted functions makes the formula weaker, and consequently may make a valid formula invalid.

Definition 3.19 (Equality logic with Uninterpreted Functions)**.** Let $\mathcal{S}$ be a given set of sort symbols. UF denotes the equality logic and uninterpreted functions defined over the signature $\text{UF} = (\mathcal{S}_{\text{UF}}, \mathcal{F}_{\text{UF}}, \mathcal{P}_{\text{UF}})$ with:

- $\mathcal{S}_{\text{UF}} = \mathcal{S}$, the given set of sort symbols;
- $\mathcal{F}_{\text{UF}} = \mathfrak{F}_{\mathcal{S}}$ a countably infinite set of fresh function symbols $f : s_1 \times \cdots \times s_n \to s$ with $s_1 \times \cdots \times s_n \in \mathcal{S}^n$ and $s \in \mathcal{S}$;
- $\mathcal{P}_{\text{UF}} = \mathfrak{P}_{\mathcal{S}} \bigcup \{=_s \mid s \in \mathcal{S}\}$, where $\mathfrak{P}_{\mathcal{S}}$ is a countably infinite set of fresh predicate symbols $p : s_1 \times \cdots \times s_n$ with $s_1 \times \cdots \times s_n \in \mathcal{S}^n$, and where $\{=_s \mid s \in \mathcal{S}\}$ is a family of equality symbols, one for each sort.

Moreover, UF axioms are those of the following axiom schemes:

- For all sort symbols $s \in \mathcal{S}_\Sigma$,

$$\begin{aligned} \forall x : s.\, (x =_s x) && \textbf{Symmetry} \\ \forall x : s.\, \forall y : s.\, (x =_s y) \Rightarrow (y =_s x) && \textbf{Reflexivity} \\ \forall x : s.\, \forall y : s.\, \forall z : s.\, (x =_s y \wedge y =_s z) \Rightarrow (x =_s z) && \textbf{Transitivity} \end{aligned}$$

- For all sort symbols $s_1 \in \mathcal{S}_\Sigma, \ldots, s_n \in \mathcal{S}_\Sigma$ and $s \in \mathcal{S}_\Sigma$, for all function symbols $f : s_1 \times \cdots \times s_n \to s \in \mathfrak{F}_{\mathcal{S}_\Sigma}$,

$$\begin{aligned} &\forall_{1 \leq i \leq n} x_i : s_i.\, \forall_{1 \leq i \leq n} y_i : s_i. && \textbf{Functional consistency} \\ &\left(\bigwedge_{1 \leq i \leq n} x_i =_{s_i} y_i\right) \Rightarrow f(x_1, \ldots, x_n) =_s f(y_1, \ldots, y_n) \end{aligned}$$

- For all sort symbols $s_1 \in \mathcal{S}_\Sigma, \ldots, s_n \in \mathcal{S}_\Sigma$ and $s \in \mathcal{S}_\Sigma$, for all predicate symbols $p : s_1 \times \cdots \times s_n \in \mathfrak{P}_{\mathcal{S}_\Sigma}$,

$$\begin{aligned} &\forall_{1 \leq i \leq n} x_i : s_i.\, \forall_{1 \leq i \leq n} y_i : s_i. && \textbf{Predicative consistency} \\ &\left(\bigwedge_{1 \leq i \leq n} x_i =_{s_i} y_i\right) \Rightarrow p(x_1, \ldots, x_n) \Rightarrow p(y_1, \ldots, y_n) \end{aligned}$$

Deciding a Conjunction of Equalities and Uninterpreted Functions Shostak introduced in 1978 [Sho78] a two-stages algorithm to decide conjunctions of equalities and uninterpreted functions. Intuitively, first we build congruence-closed equivalence classes of equal terms, and second we look for unequal terms which belong to the same equivalence class. More precisely:

1) (a) Initially, put every term in its own equivalence class.
   (b) Then, for every terms t_1 and t_2 such that $t_1 = t_2$, merge equivalence classes to which t_1 and t_2 belong.
   (c) Finally, compute the congruence closure: For every terms t_1 and t_2 belonging to the same equivalence class, for every uninterpreted function f such that $f(t_1)$ and $f(t_2)$ belong to distinct equivalence classes, merge $f(t_1)$ and $f(t_2)$ equivalence classes; Repeat until there are no more such instances.
2) If there exists a disequality $t_1 \neq t_2$ with t_1 and t_2 belonging to the same equivalence class, then the formula is unsatisfiable, else the formula is satisfiable.

Note that operations over equivalence classes can be implemented efficiently with a union-find data-structure [NO05].

As an example, let us first consider the formula $(f(x) = y) \wedge (x = f(y)) \wedge (f(x) \neq f(y))$, involving two variables x and y, and one uninterpreted function

f. We start with an equivalence class for each term: $\{x\}, \{y\}, \{f(x)\}, \{f(y)\}$. Because of $f(x) = y$ and $x = f(y)$, we merge their respective equivalence classes: $\{x, f(y)\}, \{y, f(x)\}$. As equivalence classes are already congruence-closed, we jump to the final step. Because there is a single disequality $f(x) \neq f(y)$, and because $f(x)$ and $f(y)$ belong to two different equivalence classes, we conclude the formula is satisfiable. Given two distinct values $\mathtt{a}$ and $\mathtt{b}$, an admissible model is $\{x = \mathtt{a}, y = \mathtt{b}, f(x) = \mathtt{b}, f(y) = \mathtt{a}\}$.

Let us now consider the formula $(x = y) \wedge (f(x) = y) \wedge (x \neq f(y))$. From $x = y$ and $f(x) = y$, we obtain two equivalence classes: $\{x, y, f(x)\}, \{f(y)\}$. This time, these two equivalence classes are not congruence-closed. Because $x = y$, $f(x)$ and $f(y)$ equivalence classes have to be merged, and only a single one remains: $\{x, y, f(x), f(y)\}$. Finally, $f(x) \neq f(y)$, but $f(x)$ and $f(y)$ belong to the same equivalence class, the formula is unsatisfiable.

3.3.2 Bit-Vectors

A computer system uses bit-vectors to encode information, in particular to encode numbers. Owing to the finite domain of these bit-vectors, the semantics of arithmetic operators no longer matches the one we are used to when reasoning with natural numbers, but is instead defined by means of modular arithmetic. For example, the formula $(x - y > 0) \Leftrightarrow (x > y)$ holds when x and y are interpreted as natural numbers, but no longer holds when x and y are interpreted as bit-vector, because of the possible overflow of the subtraction. Over operators, like bitwise operators, just do not have equivalent over natural numbers. For these reasons, we introduce in this section the bit-vectors theory, which can be used to correctly reason about such systems.

Definition 3.20 (Bit-Vectors)**.** The bit-vectors theory is defined over the signature $\Sigma = (\mathcal{S}_\Sigma, \mathcal{F}_\Sigma, \mathcal{P}_\Sigma)$ with:

- $\mathcal{S}_\Sigma = \{\text{BitVec}\, n \mid n \in \mathbb{N}^\star\}$ a sort symbols family for bit-vectors of size $n \in \mathbb{N}^\star$;
- $\mathcal{F}_\Sigma = \{C_n \bigcup \mathcal{U}_n \bigcup \mathcal{B}_n \mid n \in \mathbb{N}^\star\}$ where C_n is the finite set of constant bit-vectors of size n, where $\mathcal{U}_n = \left\{-_n, \sim_n, [i, j]_n\right\}$ is the set of unary operators over bit-vectors of size n, and where $\mathcal{B}_n = \{+_n, \times_n, \div_n, |_n, \&_n, \ll_n, \gg_n, \cdot_n\}$ is the set of binary operators over bit vectors of size n;
- $\mathcal{P}_\Sigma = \{<_n \mid n \in \mathbb{N}^\star\}$ a comparison symbols family over bit-vectors of size n.

The sort BitVec n is the set of finite functions from $[0, n[$ to $\{0, 1\}$. Thus, C_n is the set of the 2^n possible functions from $[0, n[$ to $\{0, 1\}$. Let us give two conversion functions,

$[\![b : \mathrm{BitVec}\, n]\!]_{\mathbb{N}} = \sum_{0 \leq i < n} b\,(i) \cdot 2^i$ which interprets bit-vectors as natural numbers, and $[\![m : \mathbb{N}]\!]_{\mathrm{BitVec}\, n} = \lambda\, i \in [0, n[\mapsto \left(m \div 2^i\right) \bmod 2$ which interprets natural numbers as bit-vectors. Then, assuming the usual interpretation of natural numbers, the bit-vector theory introduces the following axiom schemes:

- For all $n \in \mathbb{N}^\star$, for all $i \in \mathbb{N}$, for all $j \in \mathbb{N}$ such that $0 \leq i \leq j < n$,

$$\begin{array}{rr}
\forall b : \mathrm{BitVec}\, n. & \\
\quad -_n\, b = [\![2^n - [\![b]\!]_{\mathbb{N}}]\!]_{\mathrm{BitVec}\, n} & \textbf{Arithmetic negation} \\
\quad \sim_n b = \lambda\, k \in [0, n[\mapsto 1 - b\,(k) & \textbf{Logical negation} \\
b\,[i, j]_n = \lambda\, k \in [0, j - i + 1[\mapsto b\,(k + i) & \textbf{Extraction}
\end{array}$$

- For all $n \in \mathbb{N}^\star$, for all $i \in \mathbb{N}^\star$, for all $j \in \mathbb{N}^\star$ such that $i + j = n$,

$$\begin{array}{rr}
\forall b_1 : \mathrm{BitVec}\, n. \forall b_2 : \mathrm{BitVec}\, n. & \\
\quad b_1 +_n b_2 = [\![[\![b_1]\!]_{\mathbb{N}} + [\![b_2]\!]_{\mathbb{N}}]\!]_{\mathrm{BitVec}\, n} & \textbf{Addition} \\
\quad b_1 \times_n b_2 = [\![[\![b_1]\!]_{\mathbb{N}} \times [\![b_2]\!]_{\mathbb{N}}]\!]_{\mathrm{BitVec}\, n} & \textbf{Multiplication} \\
\quad [\![b_2]\!]_{\mathbb{N}} \neq 0 \Rightarrow b_1 \div_n b_2 = [\![[\![b_1]\!]_{\mathbb{N}} \div [\![b_2]\!]_{\mathbb{N}}]\!]_{\mathrm{BitVec}\, n} & \textbf{Division} \\
\quad b_1 \mid_n b_2 = \lambda\, k \in [0, n[\mapsto \max\, b_1(k)\ b_2(k) & \textbf{Logical disjunction} \\
\quad b_1 \,\&_n\, b_2 = \lambda\, k \in [0, n[\mapsto \min\, b_1(k)\ b_2(k) & \textbf{Logical conjunction} \\
\quad b_1 \ll_n b_2 = \left[\!\left[[\![b_1]\!]_{\mathbb{N}} \times 2^{[\![b_2]\!]_{\mathbb{N}}} \right]\!\right]_{\mathrm{BitVec}\, n} & \textbf{Logical shift left} \\
\quad b_1 \gg_n b_2 = \left[\!\left[[\![b_1]\!]_{\mathbb{N}} \div 2^{[\![b_2]\!]_{\mathbb{N}}} \right]\!\right]_{\mathrm{BitVec}\, n} & \textbf{Logical shift right}
\end{array}$$

$$\begin{array}{rr}
\forall b_1 : \mathrm{BitVec}\, i. \forall b_2 : \mathrm{BitVec}\, j. & \textbf{Concatenation} \\
\quad b_1 \cdot_n b_2 = \lambda\, k \in [0, n[\mapsto \begin{cases} b_1\ (k - j) & \text{when } k \geq j \\ b_2\ (k) & \text{when } k < j \end{cases} &
\end{array}$$

$$\forall b_1 : \mathrm{BitVec}\, n. \forall b_2 : \mathrm{BitVec}\, n.\, b_1 <_n b_2 \Rightarrow [\![b_1]\!]_{\mathbb{N}} < [\![b_2]\!]_{\mathbb{N}} \quad \textbf{Comparison}$$

Deciding Bit-Vector Arithmetic The most commonly used decision procedure for bit-vector arithmetic is called flattening, or bit-blasting [WSK05]. Intuitively, it reduces a bit-vector formula into a propositional formula by means of a circuit encoding where bits are propositional variables. More precisely, for every bit-vector of size n we introduce n fresh propositional variables, one for each bit. Hence, given $a : \mathrm{BitVec}\, n$, $b : \mathrm{BitVec}\, n$ and $c : \mathrm{BitVec}\, n$, we introduce the propositional variables $a_0, \ldots, a_{n-1}$, $b_0 \ldots b_{n-1}$ and $c_0, \ldots, c_{n-1}$. Then we replace bit-vector operations by logical circuits implementing them, encoded as a set of propositional constraint. Most bitwise operations are straightforward, as for example the bit-vector logical disjunction $a = b \mid_n c$, which is

replaced by propositional constraints $\bigwedge_{0 \leq i < n} a_i = b_i \vee c_i$ modelling the disjunction at the level of individual bits. Arithmetic operations are usually more complicated, as for example the bit-vector addition $a = b +_n c$ implemented with a full-adder circuit, which introduces $n - 1$ intermediate propositional variables r_i to encode carry propagation:

$$\begin{aligned} & a_0 = (b_0 \vee c_0) \wedge (\neg b_0 \vee \neg c_0) \\ \bigwedge_{1 \leq i < n} & \begin{array}{l} a_i = (\neg b_i \vee \neg c_i \vee r_i) \wedge (\neg b_i \vee c_i \vee \neg r_i) \wedge (b_i \vee \neg c_i \vee \neg r_i) \wedge (b_i \vee c_i \vee r_i) \\ r_i = (b_{i-1} \vee c_{i-1}) \wedge (b_{i-1} \vee r_{i-1}) \wedge (c_{i-1} \vee r_{i-1}) \end{array} \end{aligned}$$

For some operations like the bit-vector multiplication, circuit encodings may require to introduce thousands of fresh variables in tens of thousands new clauses, which makes the resulting propositional formula very hard to solve. For this reason, modern solvers complement bit-blasting with efficient circuit conversion [BCF+07, MV07] on one hand, and with word-level preprocessing [BDL98, GD07] and abstraction [BKO+07] on the other hand.

3.3.3 Arrays

In this section we present the arrays theory, which is used for modeling memory or data structures such as maps, vectors and hash tables. The array theory is parameterized with an index sort and an element sort, both coming with an equality relation, whose choice will affect the expressiveness of the resulting theory, and therefore its decidability. Given an index sort $\mathcal{I}$ and a element sort $\mathcal{E}$, the theory of arrays Array $\mathcal{I}$ $\mathcal{E}$ describes arrays mapping indexes $i \in \mathcal{I}$ to elements $e \in \mathcal{E}$. These arrays are defined by the two operations read $(\cdot[\cdot])$ and write $(\cdot[\cdot] \leftarrow \cdot)$, whose semantics is given by the so-called *read-over-write* axioms (ROW-axioms).

Definition 3.21 (Arrays)**.** Given a sort for indexes $\mathcal{I}$ admitting an equality relation $=_{\mathcal{I}}$, and given a sort for elements $\mathcal{E}$ admitting an equality relation $=_{\mathcal{E}}$, the theory $\mathrm{A}_{\mathcal{I}\mathcal{E}}$ of arrays mapping indexes in $\mathcal{I}$ to elements in $\mathcal{E}$ is defined over the signature $\mathrm{A}_{\mathcal{I}\mathcal{E}} = \left(\mathcal{S}_{\mathrm{A}_{\mathcal{I}\mathcal{E}}}, \mathcal{F}_{\mathrm{A}_{\mathcal{I}\mathcal{E}}}, \mathcal{P}_{\mathrm{A}_{\mathcal{I}\mathcal{E}}}\right)$ with:

- $\mathcal{S}_{\mathrm{A}_{\mathcal{I}\mathcal{E}}} = \{\mathcal{I}, \mathcal{E}, \text{Array } \mathcal{I}\ \mathcal{E}\}$, where Array $\mathcal{I}$ $\mathcal{E}$ is the sort of arrays mapping indexes in $\mathcal{I}$ to elements in $\mathcal{E}$.
- $\mathcal{F}_{\mathrm{A}_{\mathcal{I}\mathcal{E}}} = \{\cdot[\cdot], \cdot[\cdot] \leftarrow \cdot\}$ where $a[i] : \text{Array } \mathcal{I}\ \mathcal{E} \times \mathcal{I} \to \mathcal{E}$ denotes the read in array a at index i, and where $\cdot[\cdot] \leftarrow \cdot : \text{Array } \mathcal{I}\ \mathcal{E} \times \mathcal{I} \times \mathcal{E} \to \text{Array } \mathcal{I}\ \mathcal{E}$ denotes the write of element e in array a at index i.

- $\mathcal{P}_{\mathrm{A}_{\mathcal{I}\mathcal{E}}} = \left\{=_{\mathcal{I}}, =_{\mathcal{E}}, =_{\mathrm{Array}\,\mathcal{I}\,\mathcal{E}}\right\}$, where $=_{\mathrm{Array}\,\mathcal{I}\,\mathcal{E}}$: $\mathrm{Array}\,\mathcal{I}\,\mathcal{E} \times \mathrm{Array}\,\mathcal{I}\,\mathcal{E}$ denotes the equality over arrays mapping indexes in $\mathcal{I}$ to elements in $\mathcal{E}$.

Furthermore, the meaning of these new symbols is given by the following axioms:

- **Read-over-Write**

$$\forall a : \mathrm{Array}\,\mathcal{I}\,\mathcal{E}.\, \forall i : \mathcal{I}.\, \forall j : \mathcal{I}.\, \forall e : \mathcal{E}.\; (a[i] \leftarrow e)[j] = \begin{cases} e & \text{when } i =_{\mathcal{I}} j \\ a[j] & \text{otherwise} \end{cases}$$

- **Extensionality**

$$\forall a_1 : \mathrm{Array}\,\mathcal{I}\,\mathcal{E}.\, \forall a_2 : \mathrm{Array}\,\mathcal{I}\,\mathcal{E}.\; \left(a_1 =_{\mathrm{Array}\,\mathcal{I}\,\mathcal{E}} a_2\right) \Leftrightarrow (\forall i : \mathcal{I}.\, a_1[i] =_{\mathcal{E}} a_2[i])$$

Deciding the Array Theory The array theory can be reduced to a combination of index terms and element terms extended with uninterpreted functions. Note that this combination is not necessarily decidable as mentionned in Section 3.3.4. Hence the array theory can be undecidable even if the combination of index terms and element terms is decidable. However this combination is decidable for the *pure* arrays theory, i.e. when index terms and element terms are restricted to equalities.

The idea is to replace every *read-over-write* in accordance with the axiom of the same name by an **if** · **then** · **else** · construction [GD07]. We recall that f (**if** c **then** a **else** b) stands for $(c \wedge f(a)) \vee (\neg c \wedge f(b))$, or equivalently in conjunctive normal form to $(\neg c \vee f(a)) \wedge (c \vee f(b))$, where f is any function or predicate symbol. More precisely, we first turn every $(a[i] \leftarrow e)[j]$ into **if** $(i = j)$ **then** e **else** $a[j]$. In order to avoid term duplication and size explosion, sub-terms have to be shared as much as possible, and *read-over-write* can be instantiated lazily [BB09]. Then for every remaining array symbol a we introduce a fresh uninterpreted function f_a, and we replace every read $a[i]$ by the function application $f_a(i)$.

Let us consider as an example the array formula $((a[i] \leftarrow e)[j] =_{\mathcal{E}} e) \wedge (a[j] =_{\mathcal{E}} d)$. We first eliminate the write in array a by instantiating the *read-over-write* axiom: $((\textbf{if } (i =_{\mathcal{I}} j) \textbf{ then } e \textbf{ else } a[j]) =_{\mathcal{E}} e) \wedge (a[j] =_{\mathcal{E}} d)$. Then we unfold the **if** · **then** · **else** · definition: $(i \neq_{\mathcal{I}} j \vee e =_{\mathcal{E}} e) \wedge (i =_{\mathcal{I}} j \vee a[j] =_{\mathcal{E}} e) \wedge (a[j] =_{\mathcal{E}} d)$. After removing the trivial clause $(i \neq_{\mathcal{I}} j \vee e =_{\mathcal{E}} e)$, we finally turn every array symbol into uninterpreted function and substitute every read by function application: $(i =_{\mathcal{I}} j \vee f_a(j) =_{\mathcal{E}} e) \wedge (f_a(j) =_{\mathcal{E}} d)$. We end up with a formula being a combination of index terms, element terms, and uninterpreted functions.

3.3.4 Combination of Theories

The decision procedures we have studied so far focus on one specific theory. Howerer, many-sorted first-order formulas may contain terms coming from several theories. For example, the formula $(f((a[i] \leftarrow e)[g(i)]) < f(e)) \wedge (i = g(i))$ involve an array a and two uninterpreted functions f and g, and thus belong to the combination of the arrays theory with the theory of uninterpreted functions. In this section we present the Nelson-Oppen procedure [NO79, Opp80, TH96] which aims to solve the theory combination problem, and thus to obtain a solver for such combined theories. The Nelson–Oppen combination method uses equalities between variables to permit theory dedicated decision procedures to communicate information. Combined theories have to comply with several restrictions, but there are extensions to the basic Nelson–Oppen procedure which overcome each of these restrictions.

Definition 3.22 (Nelson–Oppen Restrictions)**.** In order to apply the Nelson–Oppen procedure, theories $\mathcal{T}_1, \ldots, \mathcal{T}_n$ have to comply with the following restrictions:

- $\mathcal{T}_1, \ldots, \mathcal{T}_n$ are quantifier-free first-order theories with equality;
- There is a decision procedure for each of the theories $\mathcal{T}_1, \ldots, \mathcal{T}_n$;
- The signatures of $\mathcal{T}_1, \ldots, \mathcal{T}_n$ are disjoint, they only share equality $=$;
- $\mathcal{T}_1, \ldots, \mathcal{T}_n$ are theories interpreted over an infinite domain.

Combining Convex Theories We first present a version of the Nelson-Oppen procedure which solves the theory combination problem for *convex theories*. In a convex theory, if a formula implies a disjunction of equalities, then it also implies at least one of these equalities.

Definition 3.23 (Convex Theory)**.** A Σ-theory $\mathcal{T}$ is convex if for every conjunctive Σ-formula Φ

$$\begin{array}{ll} \text{if} & \left(\Phi \Rightarrow \bigvee_{1 \leq i \leq n} x_i = y_i\right) \text{ is } \mathcal{T}\text{-valid for some } n > 1 \\ \text{then} & \left(\Phi \Rightarrow x_i = y_i\right) \text{ is } \mathcal{T}\text{-valid for some } i \in [1, n] \end{array}$$

where x_i and y_i are variables of Φ.

If some theories like the conjunctive fragment of equality logic are convex, many useful theories are nonconvex. For example, the integer linear arithmetic is nonconvex: $(x_0 = 0) \wedge (x_1 = 1) \wedge (0 \leq x_2) \wedge (x_2 \leq 1) \Rightarrow (x_0 = x_2 \vee x_1 = x_2)$ holds, but neither $(x_0 = 0) \wedge (x_1 = 1) \wedge (0 \leq x_2) \wedge (x_2 \leq 1) \Rightarrow (x_0 = x_2)$ nor $(x_0 = 0) \wedge (x_1 = 1) \wedge (0 \leq x_2) \wedge (x_2 \leq 1) \Rightarrow (x_1 = x_2)$ holds.

Nelson-Oppen Procedure The Nelson-Oppen procedure is a two-stages procedure which solves the combination problem for convex theories satisfying Nelson-Oppen restrictions:

1) The first step of the Nelson-Oppen procedure is called *purification*.

   Purification turns a formula with mixed terms into a formula where each term belongs to a specific theory by replacing every sub-term from an alien theory with a fresh variable. The formula $(f((a[i] \leftarrow e)[g(i)]) < f(e)) \wedge (i = g(i))$ mixes uninterpreted functions with arrays. Purification will turn it into $(f(x) < f(e)) \wedge (i = y) \wedge (x = (a[i] \leftarrow e)[y]) \wedge (y = g(i))$ where $f(x) < f(e)$ and $y = g(i)$ belong to the theory of uninterpreted functions, where $x = (a[i] \leftarrow e)[y]$ belongs to the arrays theory, and where $i = y$ belongs to both.

2) This new formula can thus be split into pure sub-formulas which can be solved using a decision procedure dedicated to the theory they belong to.

   (a) If one of these sub-formulas is unsatisfiable, then so is the whole formula.

   (b) Else, if a pure sub-formula implies a not already known equality between two variables, we add this equality to the formula. This process is repeated until the formula become unsatisfiable or until there is no more equality to introduce. In the latter case the formula is satisfiable.

   In our example, both sub-formulas $(f(x) < f(e)) \wedge (i = y) \wedge (y = g(i))$ and $(i = y) \wedge (x = (a[i] \leftarrow e)[y])$ are satisfiable. From the second sub-formula we learn the equality $x = e$ and propagate it to the first sub-formula. As $(f(x) < f(e)) \wedge (i = y) \wedge (y = g(i)) \wedge (x = e)$ is unsatisfiable, we conclude that the original formula is unsatisfiable.

Combining Nonconvex Theories Next, we consider the combination of nonconvex theories. We recall that it means there exist in these theories some formulas Φ such that $\Phi \Rightarrow \bigvee_{1 \leq i \leq n} x_i = y_i$, but for all i, $\Phi \not\Rightarrow x_i = y_i$. The consequence is that after propagating equalities between theories, we still have to propagate disjunctions of equalities. If a pure sub-formula implies a disjunction of equalities without implying any of these equalities, then we add the disjunction to the formula and call back the underlying DPLL procedure to split upon this disjunction.

For example with the purified formula $\phi = (f(x) < f(e)) \wedge (i = 0) \wedge (0 \leq y) \wedge (y \leq 1) \wedge (x = (a[i] \leftarrow e)[y]) \wedge (y = g(i))$ mixing uninterpreted functions, arrays and integer linear arithmetic, $(0 \leq y) \wedge (y \leq 1) \Rightarrow (y = 0 \vee y = 1)$ holds, but neither

$(0 \leq y) \wedge (y \leq 1) \Rightarrow (y = 0)$ nor $(0 \leq y) \wedge (y \leq 1) \Rightarrow (y = 1)$ holds. Therefore we add the disjunction $(y = 0 \vee y = 1)$ to the formula and call back the DPLL procedure. It results in trying to solve with the Nelson-Oppen procedure $\phi \wedge (y = 0)$ or $\phi \wedge (y = 1)$. As the second is satisfiable, we conclude to the satisfiability of the original formula.

Combining Theories Interpreted over a Finite Domain There is a rich literature on combining decision procedures for first-order theories, in particular about optimisation and extension of the Nelson-Oppen procedure. For example it is possible to extend the Nelson-Oppen procedure to theories interpreted over a finite domain, like the finite-width bit-vector theory, but in an incomplete manner. The idea is to compute a lower bound on the size of the domain in which the formula must be satisfied, and an upper bound on the number of distinct values required to satisfy the formula. For each model of the formula, we compute an upper bound on the number of distinct values it requires, and check this upper bound against the domain size lower bound. If we cannot find a model which fits into the domain, nor prove the unsatisfiability, we have to conclude we just do not know. But if we find a model fitting into the domain, then we can conclude to the satisfiability of the formula.

Beyond Nelson-Oppen Restrictions Unfortunately, the theory combination problem is undecidable for arbitrary theories. For example, the quantified Presburger arithmetic becomes undecidable when extended with uninterpreted function. Therefore there exists no general purpose combination procedure, and solver are reduced to the use of incomplete decision procedures working on a best-effort basis when dealing with undecidable theory combinations.

3.4 Conclusion

In this chapter we presented the many-sorted first-order logic, which is used by most of automatic software analyzer to reason about programs. We gave its syntax with definitions of signatures, terms and formulas, and its semantics with definitions of interpretations and models. Then we stated the decision problem of the satisfiability modulo theory, and presented a general decision procedure to solve this problem, based on a decision procedure for the propositional logic, extended first to quantifier-free formulas modulo theories, and then to quantified formulas modulo theories. As this decision procedure assume being able to solve conjunctions of atomic quantifier-free formulas in some specific theories, we finally present three theories pervasive in

software verification, the equality logic with uninterpreted functions, the bit-vectors theory and the arrays theory, together with decision procedure dedicated to their conjunctive atomic quantifier-free fragment, and a way of combining them.

Bibliography

[BB09] Robert Brummayer and Armin Biere. Lemmas on demand for the extensional theory of arrays. *JSAT*, 6(1-3):165–201, 2009.

[BCF+07] Roberto Bruttomesso, Alessandro Cimatti, Anders Franzén, Alberto Griggio, Ziyad Hanna, Alexander Nadel, Amit Palti, and Roberto Sebastiani. A lazy and layered SMT(BV) solver for hard industrial verification problems. In *Computer Aided Verification, 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007, Proceedings*, pages 547–560, 2007.

[BDL98] Clark W. Barrett, David L. Dill, and Jeremy R. Levitt. A decision procedure for bit-vector arithmetic. In *Proceedings of the 35th Conference on Design Automation, Moscone center, San Francico, California, USA, June 15-19, 1998.*, pages 522–527, 1998.

[Bjø10] Nikolaj Bjørner. Linear quantifier elimination as an abstract decision procedure. In *Automated Reasoning, 5th International Joint Conference, IJCAR 2010, Edinburgh, UK, July 16-19, 2010. Proceedings*, pages 316–330, 2010.

[BKO+07] Randal E. Bryant, Daniel Kroening, Joël Ouaknine, Sanjit A. Seshia, Ofer Strichman, and Bryan A. Brady. Deciding bit-vector arithmetic with abstraction. In *Tools and Algorithms for the Construction and Analysis of Systems, 13th International Conference, TACAS 2007, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2007 Braga, Portugal, March 24 - April 1, 2007, Proceedings*, pages 358–372, 2007.

[DLL62] Martin Davis, George Logemann, and Donald W. Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, 1962.

[DNS05] David Detlefs, Greg Nelson, and James B. Saxe. Simplify: a theorem prover for program checking. *J. ACM*, 52(3):365–473, 2005.

[DP60] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *J. ACM*, 7(3):201–215, 1960.

[GD07] Vijay Ganesh and David L. Dill. A decision procedure for bit-vectors and arrays. In *Computer Aided Verification, 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007, Proceedings*, pages 519–531, 2007.

[GHN+04] Harald Ganzinger, George Hagen, Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. DPLL(T): fast decision procedures. In *Computer Aided Verification, 16th International Conference, CAV 2004, Boston, MA, USA, July 13-17, 2004, Proceedings*, pages 175–188, 2004.

[JS97] Roberto J. Bayardo Jr. and Robert Schrag. Using CSP look-back techniques to solve real-world SAT instances. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence and Ninth Innovative Applications of Artificial Intelligence Conference, AAAI 97, IAAI 97, July 27-31, 1997, Providence, Rhode Island, USA.*, pages 203–208, 1997.

[MV07] Panagiotis Manolios and Daron Vroon. Efficient circuit to CNF conversion. In *Theory and Applications of Satisfiability Testing - SAT 2007, 10th International Conference, Lisbon, Portugal, May 28-31, 2007, Proceedings*, pages 4–9, 2007.

[NO79] Greg Nelson and Derek C. Oppen. Simplification by cooperating decision procedures. *ACM Trans. Program. Lang. Syst.*, 1(2):245–257, 1979.

[NO05] Robert Nieuwenhuis and Albert Oliveras. Proof-producing congruence closure. In *Term Rewriting and Applications, 16th International Conference, RTA 2005, Nara, Japan, April 19-21, 2005, Proceedings*, pages 453–468, 2005.

[Opp80] Derek C. Oppen. Complexity, convexity and combinations of theories. *Theor. Comput. Sci.*, 12:291–302, 1980.

[Sho78] Robert E. Shostak. An algorithm for reasoning about equality. *Commun. ACM*, 21(7):583–585, 1978.

[SS96] João P. Marques Silva and Karem A. Sakallah. GRASP - a new search algorithm for satisfiability. In *ICCAD*, pages 220–227, 1996.

[SS99] João P. Marques Silva and Karem A. Sakallah. GRASP: A search algorithm for propositional satisfiability. *IEEE Trans. Computers*, 48(5):506–521, 1999.

[TH96] Cesare Tinelli and Mehdi T. Harandi. A new correctness proof of the nelson-oppen combination procedure. In *Frontiers of Combining Systems, First International Workshop FroCoS 1996, Munich, Germany, March 26-29, 1996, Proceedings*, pages 103–119, 1996.

[Tin02] Cesare Tinelli. A DPLL-based calculus for ground satisfiability modulo theories. In *Logics in Artificial Intelligence, European Conference, JELIA 2002, Cosenza, Italy, September, 23-26, Proceedings*, pages 308–319, 2002.

[WSK05] Markus Wedler, Dominik Stoffel, and Wolfgang Kunz. Normalization at the arithmetic bit level. In *Proceedings of the 42nd Design Automation Conference, DAC 2005, San Diego, CA, USA, June 13-17, 2005*, pages 457–462, 2005.

Chapter 4

Symbolic Execution

Symbolic Execution is an automated software verification technique which had been proved successful for generating high-coverage test suites and for finding deep errors in complex software [BGM13, CS13]. If the idea of running a program symbolically was introduced in the mid 1970s [Cla76, Kin76], it has only recently been made practical, driven by significant advances in constraint solving and the development of more scalable approaches [GKS05, SMA05, WMMR05, CGP+06]. Today Symbolic Execution is integrated in industrial testing processes [GLM12] and has found several bugs in many widely used softwares [CDE08]. Also, when a tool built on Symbolic Execution won the DARPA Cyber Grand Challenge [cgc16, CARB12, ARCB14], it have been brought to the attention of a larger audience as a vulnerability detection technique.

In this chapter we formally define Symbolic Execution. Because it is about symbolically executing program, we need a language to express programs. In Section 4.1 we present the syntax and the semantics of LOW, a low-level programming language intended to mimic assembly language. Given a program in LOW, we explain in Section 4.2 how to perform Symbolic Execution on this program. In Section 4.3 we point several limits that Symbolic Execution suffers and present some solutions. Finally in Section 4.4 we show how to adapt Symbolic Execution to software verification.

4.1 LOW, a Simple Low Level Language

This section presents the syntax and the semantics of LOW, a simple low-level programming language. LOW is intended to mimic assembly like programming languages while remaining reasonably concise.

4.1.1 Syntax

Let $\mathbb{B}$ be the set of binary digits, either $\top$ or $\bot$, and let $\mathbb{M}_n$ be the set of size n machine integers, a finite range of mathematical integers. Machine integers of size n are represented as a group of n binary digits using two's complement for negative values representation.

Definition 4.1 (Syntax of LOW)**.** The LOW programming language is defined over the three following syntactic sets:

- $\mathcal{A}$ the set of arithmetic expressions:

  $$a := m \mid \textbf{get} \mid @[a] \mid a + a \mid a - a \mid a \times a \mid \ldots$$

  where $m \in \mathbb{M}_n$ denotes machine integer constants, where $@[\cdot]$ denotes in memory reads, and where **get** denotes inputs gotten from the environment.

- $\mathcal{B}$ the set of boolean expressions:

  $$b := \textbf{true} \mid \textbf{false} \mid a = a \mid a < a \mid \neg b \mid b \wedge b \mid b \vee b \mid \ldots$$

- $\mathcal{C}$ the set of commands (or instructions):

  $$c := \textbf{skip} \mid @[a] \leftarrow a \mid \textbf{if } b \textbf{ jump } a \mid \textbf{return } a \mid \textbf{abort}$$

  where $@[\cdot] \leftarrow \cdot$ denotes in memory writes, where **if** $\cdot$ **jump** $\cdot$ denotes conditional jumps, where **return** $\cdot$ denotes outputs returned to the environment, and where **abort** denotes program abortions on errors.

A LOW program is a sequence of $\mathcal{C}$ instructions indexed by an integer, the instruction position in the program code, and evaluated with regard to a readable and writable global memory. The execution of a LOW program starts on the instruction at index 0 with an initially empty memory. Instructions are executed sequentially until a conditional jump is encountered. If the jump condition is satisfied, then the execution is transferred to the instruction indicated by the jump instruction. Else the execution continues sequentially.

Figure 4.1 gives two example of LOW programs, a GCD and an integer square root computation. For example, the GCD program starts getting two integer inputs from then environment (instruction 0 and 1), writes them in memory at indexes 0 and 1, and (instruction 2) jumps to instruction 6. Then, until the value read in memory at index 1 is greater than zero (instruction 6), it will execute repeatedly instructions 3, 4 and 5. In the end, the program reaches instruction 7 and returns to the environment the value in memory at index 0.

```
0 :  @[0] ← get
1 :  @[1] ← get
2 :  if true jump 6
3 :  @[2] ← @[1]
4 :  @[1] ← @[0] mod @[1]
5 :  @[0] ← @[2]
6 :  if (0 < @[1]) jump 3
7 :  return @[0]
```

```
0 :  @[0] ← get
1 :  @[1] ← 0
2 :  @[2] ← 1
3 :  @[3] ← 0
4 :  if true jump 8
5 :  @[3] ← @[3] + @[2]
6 :  @[2] ← @[2] + 2
7 :  @[1] ← @[1] + 1
8 :  if (@[3] < @[0]) jump 5
9 :  return @[1]
```

Figure 4.1 – Euclidian algorithm for GCD (left) and integer square root computation (right) in LOW.

Language Extensions The LOW programming language we describe lacks several features usually available in most programming languages, like the presence of variables, assertions checking, or a wider variety of control-flow instructions. However all of these features can be encoded in LOW:

Variable Declarations The most blatant lack in LOW is the absence of variables, which is a commodity provided by almost all programming languages. Given a set of variable names, let us consider the language extension which adds variable assignments to LOW instructions and variable evaluations to LOW arithmetic expressions. This language extension can be reduced to LOW core syntax by associating to each variable a fixed memory location, and by replacing variable assignments by memory writes and variable evaluations by memory reads.

Assertions Many programming languages offer the possibility to check assertions at runtime and to abort the program execution in case of infringement. A language extension which adds assertions to LOW instructions can easily be encoded using conditional jumps followed by **abort** instructions.

Control-Flow Instructions The LOW programming language only provides conditional jump as control-flow instruction. But all other usual control-flow instructions like for example **goto** or **switch** can be implemented from this one, the former simply with a **if true jump** $\cdot$, the latter using a jump table.

4.1.2 Semantics

We gave an intuitive model with which to understand the behaviors of programs written in LOW. In Figure 4.2 we now give a formal definition of the LOW semantics. The operational semantics of LOW is defined against a global environment made of a map from machine integers to instructions $\Sigma : \mathbb{M}_n \to C$, the code of a LOW program, and a map from machine integers to machine integers $\Gamma : \mathbb{M}_n \to \mathbb{M}_n$, the memory accessed by a LOW program. The memory is initialised with non-deterministic values, and by convention, every unspecified code location contains an **abort** instruction.

Evaluation rules for arithmetic (resp. boolean) expressions are given in Figure 4.2a (resp. Figure 4.2b). They are expressed as relations between a memory state, an expression and a result. For example with arithmetic expressions, if **get** evaluates to m (**get** $\rightsquigarrow m$), then within memory Γ, the arithmetic expression **get** evaluates to m ($\langle \Gamma, \textbf{get} \rangle \rightsquigarrow_{\mathbb{M}_n} m$). If within memory Γ, the arithmetic expression a evaluates to i ($\langle \Gamma, a \rangle \rightsquigarrow_{\mathbb{M}_n} i$), and if memory Γ contains value m at index i ($\Gamma[i] = m$), then within memory Γ, the arithmetic expression $@[a]$ evaluates to n ($\langle \Gamma, @[a] \rangle \rightsquigarrow_{\mathbb{M}_n} m$). For example with boolean expressions, if within memory Γ arithmetic expressions a_0 and a_1 evaluate to m_0 and m_1 ($\langle \Gamma, a_0 \rangle \rightsquigarrow_{\mathbb{M}_n} m_0$ and $\langle \Gamma, a_1 \rangle \rightsquigarrow_{\mathbb{M}_n} m_1$), then within memory Γ boolean expression $a_0 < a_1$ evaluates to $m_0 < m_1$ ($\langle \Gamma, a_0 < a_1 \rangle \rightsquigarrow_{\mathbb{B}} m_0 < m_1$). Evaluation rules for instructions are given in Figure 4.2c, and are expressed as transitions between memory states, code pointers, and instructions, up to a final result. For example, if within memory Γ arithmetic expressions a_0 and a_1 evaluate to i and m ($\langle \Gamma, a_0 \rangle \rightsquigarrow_{\mathbb{M}_n} i$ and $\langle \Gamma, a_1 \rangle \rightsquigarrow_{\mathbb{M}_n} m$), then within memory Γ execution of instruction $@[a_0] \leftarrow a_1$ at code position ρ continues with execution within memory $(\Gamma[i] \leftarrow m)$ of instruction $\Sigma\,(\rho + 1)$ at code position $\rho + 1$ ($\langle \Gamma, \rho, @[a_0] \leftarrow a_1 \rangle \rightsquigarrow \langle (\Gamma[i] \leftarrow m), \rho + 1, \Sigma\,(\rho + 1) \rangle$). If within memory Γ boolean expression b evaluates to $\top$ ($\langle \Gamma, b \rangle \rightsquigarrow_{\mathbb{B}} \top$) and arithmetic expression a evaluates to i ($\langle \Gamma, a \rangle \rightsquigarrow_{\mathbb{M}_n} i$), then within memory Γ execution of instruction **if** b **jump** a continues with execution within memory Γ of instruction $\Sigma\,(i)$ at code position i ($\langle \Gamma, \rho, \textbf{if } b \textbf{ jump } a \rangle \rightsquigarrow \langle \Gamma, i, \Sigma\,(i) \rangle$). Finally, if within memory Γ arithmetic expression a evaluates to r ($\langle \Gamma, a \rangle \rightsquigarrow_{\mathbb{M}_n} r$), then if within Γ execution reaches instruction **return** a, then execution stops and returns r ($\langle \Gamma, \rho, \textbf{return } a \rangle \rightsquigarrow (\Gamma, r)$). But if within Γ execution reaches an instruction **abort**, then execution stops and fails $\lightning$ ($\langle \Gamma, \rho, \textbf{abort} \rangle \rightsquigarrow (\Gamma, \lightning)$).

4.2 Symbolic Execution

In this section we define Symbolic Execution over LOW programs. In Section 4.2.1 we give the general principle behind Symbolic Execution, with the definition of symbolic

$$\frac{}{\langle \Gamma, m\rangle \rightsquigarrow_{\mathbb{M}_n} m} \qquad \frac{\mathbf{get} \rightsquigarrow m}{\langle \Gamma, \mathbf{get}\rangle \rightsquigarrow_{\mathbb{M}_n} m}$$

$$\frac{\langle \Gamma, a\rangle \rightsquigarrow_{\mathbb{M}_n} i \qquad \Gamma[i] = m}{\langle \Gamma, @[a]\rangle \rightsquigarrow_{\mathbb{M}_n} m} \qquad \frac{\langle \Gamma, a_0\rangle \rightsquigarrow_{\mathbb{M}_n} m_0 \qquad \langle \Gamma, a_1\rangle \rightsquigarrow_{\mathbb{M}_n} m_1}{\langle \Gamma, a_0 - a_1\rangle \rightsquigarrow_{\mathbb{M}_n} m_0 - m_1}$$

(a) Concrete semantics of arithmetic expressions

$$\frac{}{\langle \Gamma, \mathbf{true}\rangle \rightsquigarrow_{\mathbb{B}} \top} \qquad \frac{}{\langle \Gamma, \mathbf{false}\rangle \rightsquigarrow_{\mathbb{B}} \bot}$$

$$\frac{\langle \Gamma, a_0\rangle \rightsquigarrow_{\mathbb{M}_n} m_0 \qquad \langle \Gamma, a_1\rangle \rightsquigarrow_{\mathbb{M}_n} m_1}{\langle \Gamma, a_0 < a_1\rangle \rightsquigarrow_{\mathbb{B}} m_0 < m_1}$$

$$\frac{\langle \Gamma, b\rangle \rightsquigarrow_{\mathbb{B}} t}{\langle \Gamma, \neg b\rangle \rightsquigarrow_{\mathbb{B}} \neg t} \qquad \frac{\langle \Gamma, b_0\rangle \rightsquigarrow_{\mathbb{B}} t_0 \qquad \langle \Gamma, b_1\rangle \rightsquigarrow_{\mathbb{B}} t_1}{\langle \Gamma, b_0 \wedge b_1\rangle \rightsquigarrow_{\mathbb{B}} t_0 \wedge t_1}$$

(b) Concrete semantics of boolean expressions

$$\frac{}{\langle \Gamma, \rho, \mathbf{skip}\rangle \rightsquigarrow \langle \Gamma, \rho + 1, \Sigma\,(\rho + 1)\rangle}$$

$$\frac{\langle \Gamma, a_0\rangle \rightsquigarrow_{\mathbb{M}_n} i \qquad \langle \Gamma, a_1\rangle \rightsquigarrow_{\mathbb{M}_n} m}{\langle \Gamma, \rho, @[a_0] \leftarrow a_1\rangle \rightsquigarrow \langle (\Gamma[i] \leftarrow m), \rho + 1, \Sigma\,(\rho + 1)\rangle}$$

$$\frac{\langle \Gamma, b\rangle \rightsquigarrow_{\mathbb{B}} \top \qquad \langle \Gamma, a\rangle \rightsquigarrow_{\mathbb{M}_n} i}{\langle \Gamma, \rho, \mathbf{if}\ b\ \mathbf{jump}\ a\rangle \rightsquigarrow \langle \Gamma, i, \Sigma\,(i)\rangle}$$

$$\frac{\langle \Gamma, b\rangle \rightsquigarrow_{\mathbb{B}} \bot}{\langle \Gamma, \rho, \mathbf{if}\ b\ \mathbf{jump}\ a\rangle \rightsquigarrow \langle \Gamma, \rho + 1, \Sigma\,(\rho + 1)\rangle}$$

$$\frac{\langle \Gamma, a\rangle \rightsquigarrow_{\mathbb{M}_n} r}{\langle \Gamma, \rho, \mathbf{return}\ a\rangle \rightsquigarrow (\Gamma, r)} \qquad \frac{}{\langle \Gamma, \rho, \mathbf{abort}\rangle \rightsquigarrow (\Gamma, \lightning)}$$

(c) Concrete semantics of instructions

Figure 4.2 – Concrete semantics of LOW.

expressions, symbolic states and path constraints. Then in Section 4.2.2 we explore some advanced Symbolic Execution techniques used in modern tools in order to scale over large programs.

4.2.1 General Principle

The main idea behind Symbolic Execution is to replace inputs with symbols that can take any value, and, as the name of the technique suggests, to execute the program over these symbols. Symbolic Execution explores the program by enumerating all the possible execution paths in the program control-flow graph. All along each of these paths, it maintains: 1) A *symbolic state* that maps variables to symbolic expressions or values; 2) A *path constraint*, also known as *path predicate*, a usually quantifier-free first-order formula over symbolic expressions, which describes the constraints that symbolic inputs have to satisfy in order to follow the branches taken by a path. Branch execution updates the path constraint, while assignments update the symbolic store. When Symbolic Execution reaches the end of a path, a constraint solver — typically a SMT solver — is used to solve the path constraint and to compute a set of inputs making the execution to follow that path.

In other words, rather than taking concrete inputs values and exploring the path followed by the execution on these inputs, Symbolic Execution abstractly represents inputs as symbols, chooses a path to explore, and resorts to constraint solvers to construct actual instances that follow that path.

Symbolic Expression and Symbolic State Symbolic Execution uses symbolic values instead of concrete data values as input, and represents data values computed by the program as symbolic expressions over these symbolic input values. As their name suggests, symbolic values are logical symbols while symbolic expressions are logical terms, both interpreted within the theory which was chosen to model program values. For LOW programs, we choose the bit-vectors theory for arithmetic symbolic values, denoted by BitVec n and presented in Section 3.3.2, and the boolean theory for boolean symbolic values, denoted by Bool. Symbolic expressions are evaluated against a symbolic state Γ, which maps program variables to symbolic expressions. As there is no variable declaration in LOW, this symbolic state will, in our specific case, only contain implicitly declared memory variables, which are mapped to symbolic expressions interpreted within the arrays theory presented in Section 3.3.3.

LOW expressions are turned into first-order arrays bit-vectors symbolic expressions according to the translation given in Figure 4.3a. Constant machine integers are

$$\frac{}{\langle \Gamma, m\rangle \twoheadrightarrow_{\text{BitVec}\,n} m} \qquad \frac{v : \text{BitVec}\,n \text{ a fresh symbol}}{\langle \Gamma, \mathbf{get}\rangle \twoheadrightarrow_{\text{BitVec}\,n} v}$$

$$\frac{\langle \Gamma, a\rangle \twoheadrightarrow_{\text{BitVec}\,n} i}{\langle \Gamma, @[a]\rangle \twoheadrightarrow_{\text{BitVec}\,n} \Gamma[i]} \qquad \frac{\langle \Gamma, a_0\rangle \twoheadrightarrow_{\text{BitVec}\,n} m_0 \qquad \langle \Gamma, a_1\rangle \twoheadrightarrow_{\text{BitVec}\,n} m_1}{\langle \Gamma, a_0 - a_1\rangle \twoheadrightarrow_{\text{BitVec}\,n} m_0 -_n m_1}$$

(a) Symbolic semantics of arithmetic expressions

$$\frac{}{\langle \Gamma, \mathbf{true}\rangle \twoheadrightarrow_{\text{Bool}} \top} \qquad \frac{}{\langle \Gamma, \mathbf{false}\rangle \twoheadrightarrow_{\text{Bool}} \bot}$$

$$\frac{\langle \Gamma, a_0\rangle \twoheadrightarrow_{\text{BitVec}\,n} m_0 \qquad \langle \Gamma, a_1\rangle \twoheadrightarrow_{\text{BitVec}\,n} m_1}{\langle \Gamma, a_0 < a_1\rangle \twoheadrightarrow_{\text{Bool}} m_0 <_n m_1}$$

$$\frac{\langle \Gamma, b\rangle \twoheadrightarrow_{\text{Bool}} t}{\langle \Gamma, \neg b\rangle \twoheadrightarrow_{\text{Bool}} \neg t} \qquad \frac{\langle \Gamma, b_0\rangle \twoheadrightarrow_{\text{Bool}} t_0 \qquad \langle \Gamma, b_1\rangle \twoheadrightarrow_{\text{Bool}} t_1}{\langle \Gamma, b_0 \wedge b_1\rangle \twoheadrightarrow_{\text{Bool}} t_0 \wedge t_1}$$

(b) Symbolic semantics of boolean expressions

$$\frac{}{\langle \text{PC}, \Gamma, \rho, \mathbf{skip}\rangle \twoheadrightarrow \langle \text{PC}, \Gamma, \rho + 1, \Sigma\,(\rho + 1)\rangle}$$

$$\frac{\langle \Gamma, a_0\rangle \twoheadrightarrow_{\text{BitVec}\,n} i \qquad \langle \Gamma, a_1\rangle \twoheadrightarrow_{\text{BitVec}\,n} m}{\langle \text{PC}, \Gamma, \rho, @[a_0] \leftarrow a_1\rangle \twoheadrightarrow \langle \text{PC}, \Gamma[i] \leftarrow m, \rho + 1, \Sigma\,(\rho + 1)\rangle}$$

$$\frac{\langle \Gamma, b\rangle \twoheadrightarrow_{\text{Bool}} t \qquad \langle \Gamma, a\rangle \twoheadrightarrow_{\text{BitVec}\,n} i \qquad \text{for all } j : \text{BitVec}\,n}{\langle \text{PC}, \Gamma, \rho, \mathbf{if}\ b\ \mathbf{jump}\ a\rangle \twoheadrightarrow \langle \text{PC} \wedge t \wedge j = i, \Gamma, j, \Sigma\,(j)\rangle}$$

$$\frac{\langle \Gamma, b\rangle \twoheadrightarrow_{\text{Bool}} t}{\langle \text{PC}, \Gamma, \rho, \mathbf{if}\ b\ \mathbf{jump}\ a\rangle \twoheadrightarrow \langle \text{PC} \wedge \neg t, \Gamma, \rho + 1, \Sigma\,(\rho + 1)\rangle}$$

$$\frac{\langle \Gamma, a\rangle \twoheadrightarrow_{\text{BitVec}\,n} m \qquad \text{for all } r : \text{BitVec}\,n}{\langle \text{PC}, \Gamma, \rho, \mathbf{return}\ a\rangle \twoheadrightarrow \{(\mathcal{M}, r) \mid \mathcal{M} \models \text{PC} \wedge r = m\}}$$

$$\frac{}{\langle \text{PC}, \Gamma, \rho, \mathbf{abort}\rangle \twoheadrightarrow \{(\mathcal{M}, \lightning) \mid \mathcal{M} \models \text{PC}\}}$$

(c) Symbolic semantics of instructions

Figure 4.3 – Symbolic semantics of LOW.

converted into bit-vectors constant values, and arithmetic machine operators are lifted to their bit-vectors counterparts. Every **get** occurrence is replaced by a fresh bit-vector symbol, and in memory reads $@[\cdot]$ are replaced by symbolic reads $\cdot[\cdot]$ over the symbolic expression representing the current state of the memory. The translation of boolean expression is straightforward, as all boolean machine values and all boolean machine operators can be directly lifted to their counterparts in the boolean theory, following rules given in Figure 4.3b.

Path Constraints Beside the symbolic state Γ, Symbolic Execution maintains a path constraint PC, which is a quantifier-free first-order formula over symbolic expressions. PC gathers constraints that symbolic inputs have to satisfy in order to ensure the execution follows a specific path in the program control flow.

Symbolic Execution computes for each path a set of inputs for which the concrete execution follows the same path. Figure 4.3c details how these sets of inputs can be computed following transition rules between states $\langle \text{PC}, \Gamma, \rho, \Sigma(\rho) \rangle$, where PC contains path constraints and Γ is the symbolic state. Symbolic Execution starts in the initial state $\langle \top, \gamma, 0, \Sigma(0) \rangle$, with γ a fresh array symbol. On a write instruction $@[a_0] \leftarrow a_1$, the symbolic state is updated into $\Gamma[i] \leftarrow m$, where $\langle \Gamma, a_0 \rangle \twoheadrightarrow_{\text{BitVec}\, n} i$ and $\langle \Gamma, a_1 \rangle \twoheadrightarrow_{\text{BitVec}\, n} m$ are bit-vector symbolic translations of a_0 and a_1 according to Γ. Note that $\cdot[\cdot] \leftarrow \cdot$ denotes the write of the arrays theory. Jump instruction **if** b **jump** a are more complex to handle. On these instructions, Symbolic Execution forks on all possible instructions reachable from this one, and returns the union of these forks results. More precisely, assuming $\langle \Gamma, b \rangle \twoheadrightarrow_{\text{Bool}} t$ and $\langle \Gamma, a \rangle \twoheadrightarrow_{\text{BitVec}\, n} i$, if the symbolic jump condition t can be validated, then the Symbolic Execution forks on every instruction whose code pointer j might be equal to i. If so, the path constraint becomes $\text{PC} \wedge t \wedge i = j$. Moreover, if the symbolic jump condition can be contradicted, then the Symbolic Execution continue on the next instruction, and the path constraint becomes $\text{PC} \wedge \neg t$. Symbolic Execution stops when it reaches a **return** a or an **abort** instruction. On the former, assuming $\langle \Gamma, a \rangle \twoheadrightarrow_{\text{BitVec}\, n} m$, it returns a set of couples $(\mathcal{M}, r)$, where $\mathcal{M}$ is a model of $\text{PC} \wedge r = m$, i.e. is an input on which the concrete execution follows the symbolic path and returns the value r. On the latter, it returns a set of couples $(\mathcal{M}, \lightning)$, where $\mathcal{M}$ is a model of PC, i.e. is an input on which the concrete execution follows the symbolic path and fails.

Running Example For the sake of clarity, let us symbolically execute the LOW program given in Figure 4.4. Basically, this program retrieves two integer inputs from the user, computes which one is the greatest of the two, checks if this value is greater than at least one of the two user inputs and returns it. However, the final check can be invalidated,

0 : @[0] ← **get**
1 : @[1] ← **get**
2 : **if** (@[0] < @[1]) **jump** 5
3 : @[2] ← @[0]
4 : **if true jump** 6
5 : @[2] ← @[1]
6 : **if** (@[0] < @[2] ∨ @[1] < @[2]) **jump** 8
7 : **abort**
8 : **return** @[2]

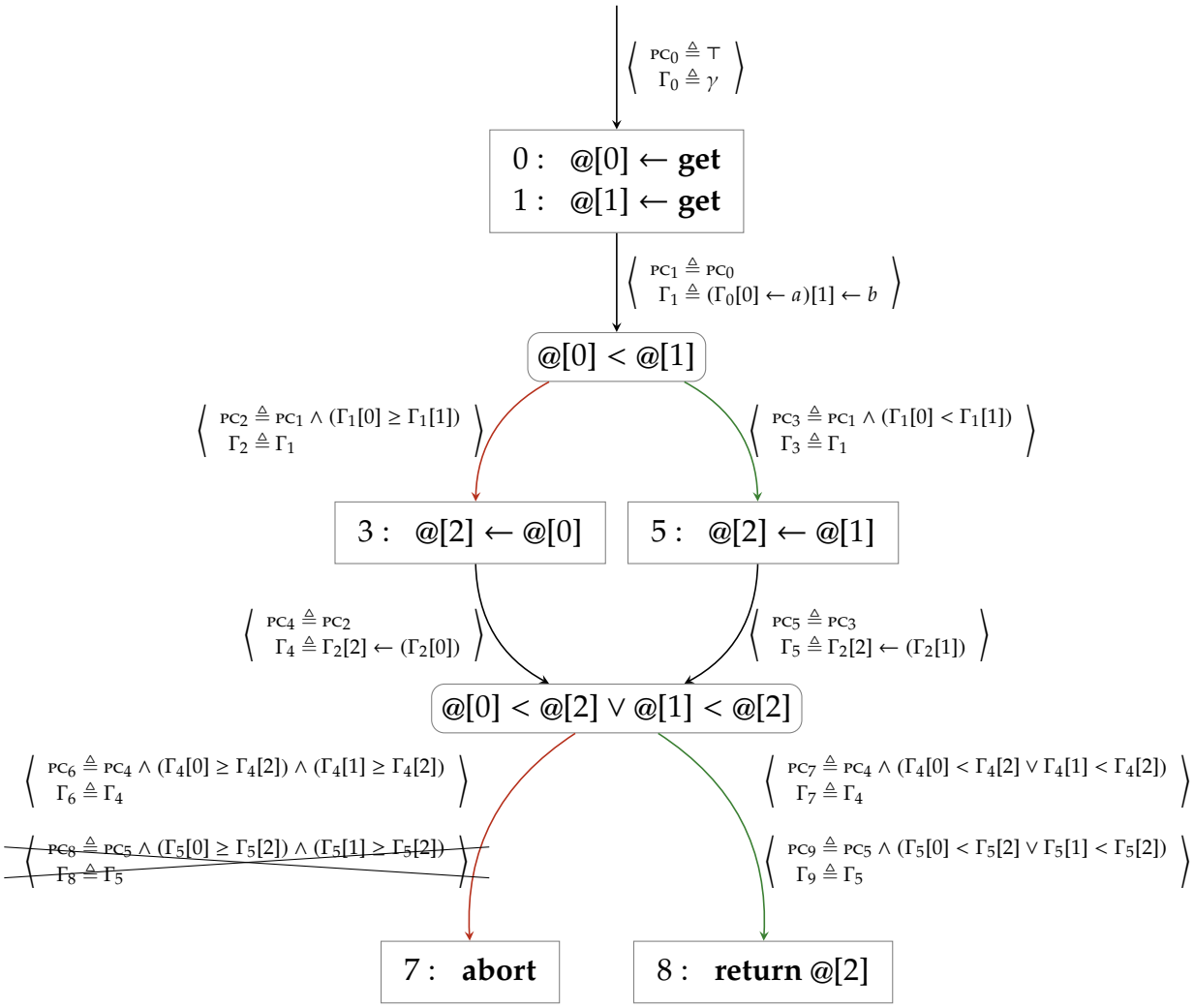


Figure 4.4 – Symbolic Execution of a LOW program.

as the Symbolic Execution will highlight it.

Symbolic Execution starts in the initial state $\text{PC}_0 \triangleq \top$, $\Gamma_0 \triangleq \gamma$, where γ is a fresh array symbol. Instructions 0 and 1 retrieve two user inputs and store them in memory at indexes 0 and 1. Therefore the symbolic state becomes $\Gamma_1 \triangleq (\Gamma_0[0] \leftarrow a)[1] \leftarrow b$, where a and b are fresh bit-vector symbols representing symbolic inputs obtained from **get** expressions. Instruction 3 is a jump instruction, consequently Symbolic Execution forks depending on the symbolic evaluation of the jump condition $@[0] < @[1]$.

On the branch where the jump condition is satisfied the path constraint becomes $\text{PC}_3 \triangleq \text{PC}_1 \wedge (\Gamma_1[0] < \Gamma_1[1])$ and the symbolic state $\Gamma_5 \triangleq \Gamma_2[2] \leftarrow (\Gamma_2[1])$ after symbolically executing instruction 5. Then the Symbolic Execution reaches the jump instruction 6. Satisfying the symbolic translation of $@[0] < @[2] \vee @[1] < @[2]$ leads to instruction 8 with path constraint $\text{PC}_9 \triangleq \text{PC}_5 \wedge (\Gamma_5[0] < \Gamma_5[2] \vee \Gamma_5[1] < \Gamma_5[2])$, where Symbolic Execution stops and returns for example $\{a = 0, b = 1\}$ However, invalidating the symbolic condition would lead to instruction 7 with path constraint $\text{PC}_8 \triangleq \text{PC}_5 \wedge (\Gamma_5[0] \geq \Gamma_5[2]) \wedge (\Gamma_5[1] \geq \Gamma_5[2])$, which is unsatisfiable. Hence it is impossible to reach the **abort** instruction from this branch.

On the branch where the jump condition is not satisfied the path constraint becomes $\text{PC}_2 \triangleq \text{PC}_1 \wedge (\Gamma_1[0] \geq \Gamma_1[1])$ and the symbolic state $\Gamma_4 \triangleq \Gamma_2[2] \leftarrow (\Gamma_2[0])$ after symbolically executing instruction 3. Once again the Symbolic Execution reaches the jump instruction 6. Satisfying the symbolic condition leads to instruction 8 with path constraint $\text{PC}_7 \triangleq \text{PC}_4 \wedge (\Gamma_4[0] < \Gamma_4[2] \vee \Gamma_4[1] < \Gamma_4[2])$, where Symbolic Execution stops and returns for example $\{a = 1, b = 0\}$. But on this branch, it is also possible to invalidate the jump condition which leads to instruction 7 with path constraint $\text{PC}_6 \triangleq \text{PC}_4 \wedge (\Gamma_4[0] \geq \Gamma_4[2]) \wedge (\Gamma_4[1] \geq \Gamma_4[2])$. Hence the Symbolic Execution reaches the **abort** instruction, fails and returns for example $\{a = 0, b = 0\}$. Indeed, providing two equal inputs systematically leads to the **abort** instruction and make the concrete execution fail.

4.2.2 Advanced Techniques

Modern Symbolic Execution tools do not exactly implement the general Symbolic Execution we presented so far, as it scales badly and has difficulties to manage externalities, but variations instead. In this section we present two advanced techniques widely used by Symbolic Execution tools, concretization and symbolization, and two variations around this general Symbolic Execution, which reflect more accurately what is implemented in modern tools [GKS05, SMA05, CKC12].

Concretization and Symbolization Modern Symbolic Execution tools do not symbolically evaluate entire execution traces, but only trace fragments in order to scale on large program [GKS05]. Concretization and symbolization are two common strategies allowing to properly achieve this goal [DBT+16]:

Concretization Concretization under-approximates the path constraint by instantiating logical symbols with concrete values, allowing to simplify or even cut off some trace fragments. These concrete values may come from run-time values of a concrete execution of the program, or from a partial solution of the path constraint. For example, we can concretize read and write indexes in order to reduce the complexity of the path constraint. We can also execute system calls and concretize corresponding trace fragments with values they return.

Symbolization Symbolization over-approximates the path constraint by abstracting some trace fragments with fresh logical symbols. This is particularly interesting when we cannot or do not want to execute the program. For example, as concretization requires to execute system calls, we may instead prefer to symbolize them with fresh logical symbols. We can also simplify the path constraint by symbolizing fragments which are computationally too hard to solve, as for example those coming from cryptographic libraries.

As concretization under-approximates and symbolization over-approximates the path constraint, one has to be very careful when mixing these two strategies as an improper combination would make the analysis unsound [DBF+16].

Static and Dynamic Symbolic Execution Instead of the general Symbolic Execution we presented so far, Symbolic Execution tools implement variations which can roughly be classified between the following two:

Static Symbolic Execution Static Symbolic Execution is the closest variant to our general Symbolic Execution [WMMR05, CDE08]. It starts by symbolically executing the program from its entry point. On a branching instruction, it solves the path constraint in order to find all the possible instructions where the execution can continue. Choosing where to continue between these destinations is a matter of heuristics and greatly influence the overall efficiency. As its name suggests, the program is never concretely executed, which make the use of run-time values impossible for concretization. Even if it is still possible to use partial solution values in order to concretize parts of the path constraint, this limitation tends to

make symbolization a more natural choice, especially when dealing with external calls.

Dynamic Symbolic Execution By contrast, dynamic Symbolic Execution relies on the ability to execute and to inspect run-time values of the program [GKS05, SMA05]. It starts by concretely executing the program on predetermined concrete initial inputs. Then these concrete inputs are symbolized, and for each branching instruction in the execution trace, it tries to find inputs for which the execution takes the direction not followed by the initial trace. The process is repeated on each of these new concrete inputs. As the exploration starts with a concrete trace, concretization is a natural choice to deal with external calls or to simplify parts of the path constraint. Symbolization can still be useful to generalize the path constraint.

Of course, modern Symbolic Execution tools are not restricted to pure static or pure dynamic Symbolic Execution, but implement something in between, depending on their verification goals and the kind of program they target.

4.3 Limits and Solutions

Despite being a mature technology, Symbolic Execution is still an active research area. The reason is that it still suffers from fundamental limitations, all of which being challenging research topics. In this section we discuss some of these challenges, path explosion in Section 4.3.1, constraint solving in Section 4.3.2, memory model in Section 4.3.3, and interaction with the environment in Section 4.3.4, together with interesting solutions developed in response to them.

4.3.1 Path Explosion

The main limitation to Symbolic Execution comes from the huge number of program paths in all but the smallest programs. Indeed, the number of program paths is at least exponential in the number of branching instructions, and can even be infinite in the presence of unbounded loop iterations. Several approaches were developed to overcome this issue:

Path Pruning A first natural strategy to reduce the path space is to invoke the constraint solver at each branching instruction to prune unrealizable paths: if the branching condition is unsatisfiable, then we can discard the path following this branch. This

approach is the default in most Symbolic Execution tools. An orthogonal approach consists in extracting an unsat core from each path proved to be unsatisfiable, and to discard all paths sharing this minimal unsat core [SSJ+15, DBG10]. We can also remove redundant paths during the exploration by discarding those reaching an already visited program point with a more constrained symbolic state than before [BCE08].

Path Merging Path merging is a powerful technique that merges different paths reaching a common location into a single one [CDE08, ARCB14]. It merges symbolic states by mapping variables to the disjunction of their symbolic representations in the different symbolic states, whereas the merged path constraint is the disjunction of the different incoming path constraints. However, path merging introduces disjunctions, which increase the complexity of constraints and put a burden on constraints solvers. Therefore there is a trade-off to be made between merging and not merging, which is a matter of heuristics [KKBC12].

Function and Loop Summarization When a function or a loop is traversed several times, we can try to build a summary of this code fragment for subsequent reuse [God07, GL11, BBKK12]. However, computing function or loop summaries is known to be hard, and is an active research topic. Summaries can still be computed under some specific conditions, for example when the body of a loop or function is restricted to linear arithmetic operation over static variables [dOBP16].

Program Analysis and Compiler Optimizations Various static analysis and compiler optimizations can be adapted to Symbolic Execution in order to gain a better understanding of the program behaviors and to ease its exploration [Cad15]. For example we can use program slicing to extract a subset of the program whose analysis is sufficient to reach a targeted program point [Wei81, Tip95, CKGJ11]. We can also use taint analysis to track dependencies between program variables and remove the parts of the program that do not depend on some chosen inputs [NS05, SAB10]. And many compiler optimizations can be applied on execution traces in order to significantly reduce their size.

4.3.2 Constraint Solving

Despite significant advances which made Symbolic Execution possible in the first place, constraint solving continues to be a bottleneck in Symbolic Execution. Indeed, without proper optimization, Symbolic Execution ends up generating queries which blow the

solver up. In particular, Symbolic Execution engines have to be cautious about the following aspects in order to prevent solver explosion:

Solution Reuse Reusing previously computed constraint results to speed up the analysis can be particularly effective in the setting of a Symbolic Execution. Indeed, because of the path based nature of Symbolic Execution, many constraints will repeatedly be sent unchanged to the solver. Most reuse approaches for constraint solving are currently based on semantic or syntactic equivalence of the constraints [CGP+06, CDE08], which makes solution reuse especially efficient when combined with other constraint simplification techniques.

Formula Size As it is the case in other unfolding-based verification techniques — like bounded model checking, Symbolic Execution produces larger and larger constraints as the path exploration progresses. Combine with extra complexity due to disjunctions introduced explicitly by path merging or implicitly by terms from the arrays theory, it results in constraints that exhaust all modern solvers. To prevent this bottleneck, it is therefore crucial to simplify these constraints directly into the Symbolic Execution engine using dedicated treatments [GD07, DBT+16].

Quantified Formulas Properly modeling program components, summarizing loop or formalizing some properties to verify, might require the use of quantification in constraints [SST13]. However, finding a model for a constraint involving universally quantification is difficult, and even undecidable for most theories handled by solvers. Despite recent advances in the resolution of quantified constraints, most solvers will fail to solve such constraints, especially the large ones produced by Symbolic Execution. Thus, quantifications have to be specifically handled in order to not burden the underlying solver.

4.3.3 Memory Model

A crucial aspect of Symbolic Execution is how to model memory in order to correctly handle pointers. One can choose a low-level memory model, like we do when we define Symbolic Execution for LOW programs, or higher-level memory models when dealing with higher level languages. Whatever the memory model we choose, we have to take into consideration the fact that the program execution starts in a widely undefined memory state. A memory model is an important design choice for Symbolic Execution tools, as it significantly affects program behaviors coverage and constraint solving scalability: too restricted memory model will induce weak program behaviors coverage while too generic memory model will exhaust solvers.

Fully Symbolic Memory Fully symbolic memory is the memory model with the highest level of generality [SBY+08, BJAS11]. Both memory addresses and memory contents are symbolic expressions, making possible to accurately describe all possible memory manipulations. However, these symbolic expressions are likely to grow quickly and to become complex to solve. Moreover, an unconstrained symbolic address will reference any memory cell, leading to an explosion in the number of program states.

Address Concretization Address concretization is a popular technique, which consists in concretizing symbolic memory addresses into single concrete addresses by replacing them with runtime values [GKS05, SMA05]. This reduces the number of states and the complexity of constraints fed to the solver, although it causes the Symbolic Execution to miss paths that depend on specific memory addresses.

4.3.4 Interactions with the Environment

As for others software verification techniques, environment interactions are sources of troubles in Symbolic Execution. Because most programs are not self-contained, Symbolic Execution has to take into account their interactions with the surrounding environment. Typical examples of this are the underlying operating system, with its file system, environment variables, and network access, and complex user interaction.

System Environment Early works dealt with the system environment by actually executing external calls using concrete arguments [GKS05, SMA05, CGP+06]. In addition to the risk of performing system calls on arbitrary values, it may also break the analysis consistency. Indeed, as there is no mechanism for tracking the external call side effects, it may result in having external calls from distinct paths interfere with each other, leading to inconsistency in the analysis. The usual way to overcome this problem is to create abstract models of system interactions; but creating such a model for all system interactions is an endless and error prone task.

User Interaction Dealing with user interaction is even worse. Indeed, concretization is not possible as it would require to trigger the user each time the program needs to interact with him. The only solution is therefore to define dedicated models, which can be extremely convoluted, for example when dealing with graphical interfaces.

4.4 Application to Program Verification

Finally we explain how to adapt Symbolic Execution to program verification. We focus in Section 4.4.1 on the verification of a specific kind of properties called *trace properties*, whose verification fits well with Symbolic Execution. Then in Section 4.4.2 we define the notions of over-approximation and under-approximation, completeness and correctness for Symbolic Execution, and express the link between these different notions.

4.4.1 Trace Property Verification

Trace properties are a specific kind of properties, stated over execution traces of the program to be analyzed. Because it performs a trace-based exploration of the program control-flow graph, Symbolic Execution happen to be well adapted for verifying such properties.

Definition 4.2 (Execution Trace)**.** An execution trace is a sequence of states that could successively be taken by a program during one of its execution.

Hence for LOW programs, concrete execution traces are members of the family $(\langle \Gamma_n, \rho_n, \Sigma(\rho_n)\rangle)_{n\in\mathbb{N}}$, where for all $n \in \mathbb{N}$ there exists a transition rule in Figure 4.2c such that $\langle \Gamma_n, \rho_n, \Sigma(\rho_n)\rangle \rightsquigarrow \langle \Gamma_{n+1}, \rho_{n+1}, \Sigma(\rho_{n+1})\rangle$, while Symbolic Execution traces are members of the family $(\langle \mathrm{pc}_n, \Gamma_n, \rho_n, \Sigma(\rho_n)\rangle)_{n\in\mathbb{N}}$, where for all $n \in \mathbb{N}$ there exists a transition rule in Figure 4.3c such that $\langle \mathrm{pc}_n, \Gamma_n, \rho_n, \Sigma(\rho_n)\rangle \rightsquigarrow \langle \mathrm{pc}_{n+1}, \Gamma_{n+1}, \rho_{n+1}, \Sigma(\rho_{n+1})\rangle$.

Definition 4.3 (Trace Property)**.** Let $\mathbb{T}$ be the set of all execution traces. A property Φ is a trace property if it can be expressed as a computable function $f_\Phi : \mathbb{T} \mapsto \mathbb{B}$ mapping execution traces to $\top$ or $\bot$. We said that Φ holds on a trace $\mathcal{T} \in \mathbb{T}$ if $f_\Phi(\mathcal{T}) = \top$, and we said that Φ holds on a program P if Φ holds on all execution traces of P.

Despite the restriction of considering only trace properties, a rich number of interesting properties can still be expressed in this way. For example, we can check that no integer overflow occurs by implementing a function which checks for every suspicious operator if it does not overflow on current operands. Or we can check that no memory address is read before it was written by implementing a function which keeps a record of every written address and verify that every read address belongs to this record.

Verification by Symbolic Execution From these definitions, adapting Symbolic Execution to trace properties verification is straightforward. We first express the property

as a function over symbolic traces, then we run the Symbolic Execution engine in order to retrieve a set of symbolic traces covering all the possible executions of the program, and finally call the function on each of these symbolic traces. The property holds on the program if it holds on each of the symbolic traces provided by the Symbolic Execution engine. Note that the set of symbolic traces for a program might be infinite. In this case, verification by Symbolic Execution will in general by incomplete, and the approach we described provides a pseudo-algorithm that may not terminate.

An alternative approach relies on the fact that any trace property can be enforced on a program by the mean of instrumentation [CF00]. The resulting instrumented program behaves exactly as the original program, except it will fail and abort if the enforced property is infringed. Having made this observation, the problem of verifying if a property holds on the original program is reduced to a reachability problem in the instrumented program. Indeed, we simply have to let the Symbolic Execution engine enumerate all the possible execution paths of the instrumented program and check that none of them leads to an **abort** instruction. *For this reason, many notions like completeness and correctness of an analysis by Symbolic Execution can be stated in terms of reachability.*

4.4.2 Correctness and Completeness

Correctness (also known as soundness) and completeness are two dual meta-properties of verifiers commonly used to measure their usefulness. A verifier is a program which is capable of checking whether a given program satisfies a given assertion. Intuitively, a verifier is correct if properties it can prove hold with respect to the semantics of the system. Conversely, a verifier is complete if it can prove all the properties which hold with respect to the semantics of the system.

Definition 4.4 (Correctness)**.** A verifier is correct if, for every program P, for every property Φ, if the verifier states that property Φ holds on program P, then Φ indeed holds on P.

Definition 4.5 (Completeness)**.** A verifier is complete if, for every program P, for every property Φ, if property Φ indeed holds on program P, then the verifier will state that Φ holds on P.

We do not consider analyzers which are incorrect and incomplete, as they are not relevant in the context of formal verification. Conversely, it is in general impossible for an analyzer to be both correct and complete, the choice of being one or the other depending on the kind of performed analysis.

Under-Approximation and Over-Approximation Notions of correctness and completeness are tightly linked to notions of under-approximation and over-approximation. If we speak in terms of program behavior, an analyzer aims to capture all the behaviors of a program and to check that properties to verify hold on these behaviors. However, capturing the exact set of behaviors for a given program is impossible. Hence, in order to make verification possible, analyzers under-approximate or over-approximate this set of behaviors. In the case of Symbolic Execution, program behaviors are materialized by execution traces, which gives us the two following definitions.

Definition 4.6 (Under-Approximation)**.** A Symbolic Execution under-approximates the concrete execution of a program if, for every feasible execution path, for every input in the set of inputs computed by the Symbolic Execution for that path, this input makes the concrete execution follow that path.

Definition 4.7 (Over-Approximation)**.** A Symbolic Execution over-approximates the concrete execution of a program if, for every feasible execution path, for every input making the concrete execution follow that path, this input belongs to the set of inputs computed by the Symbolic Execution for that path.

In the case on an under-approximating Symbolic Execution, all symbolic traces are contained in the set of concrete traces. Therefore if a property is valid on a symbolic trace, this property will be valid on a concrete trace: under-approximating Symbolic Executions are correct. Conversely, in the case of an over-approximating Symbolic Execution, all concrete traces are contained in the set of symbolic traces. Therefore if a property is valid on a concrete trace, this property will be valid on a symbolic trace: over-approximating Symbolic Executions are complete.

Proposition 4.1 (Under-Approximations are Correct)**.** *If a Symbolic Execution under-approximates the concrete execution of a program* P*, then for every trace property* Φ*, if Symbolic Execution states that* Φ *holds on a trace of* P*, then* Φ *indeed holds on that trace.*

Proposition 4.2 (Over-Approximations are Complete)**.** *If a Symbolic Execution over-approximates the concrete execution of a program* P*, then for every trace property* Φ*, if* Φ *holds on a trace of* P*, then Symbolic Execution will state that* Φ *holds on that trace.*

Under-approximation and over-approximation are both possible to achieve in Symbolic Execution, thanks, among other techniques, to concretization and symbolization, as said in Section 4.2.2. However, under-approximation tends to be a more natural choice. Indeed, in most programs, the number of distinct execution traces is huge, even sometime infinite. Symbolic Execution has to consider only a fraction of them,

and therefore under-approximates the different behaviors realizable by a program. A common workaround is to consider a weak form of completeness, a completeness *up to k*, where only the k first execution steps are considered, which makes the set of execution traces finite by bounding the size of execution traces to k.

4.5 Conclusion

In this chapter we presented Symbolic Execution, an automated software verification technique which had been proved successful in bug finding. We first introduced and gave the syntax and semantics of the LOW language, a low-level programming language intended to mimic assembly language, before explaining the general principle behind Symbolic Execution and formally defining the Symbolic Execution of LOW programs. We also detailed two advanced techniques commonly encountered in Symbolic Execution tools: concretization and symbolization; and two variations around the general Symbolic Execution: Static Symbolic Execution and Dynamic Symbolic Execution. Then we listed several limitations and challenges that Symbolic Execution tools have to face, like path explosion, memory modelling, interaction with the environment and constraint solving issues, together with approaches developed to overcome these issues. Finally we explained how to adapt Symbolic Execution to the verification of trace properties, defined the correctness and completeness meta-properties and linked these notions to the concept of under-approximating and over-approximating Symbolic Execution.

Bibliography

[ARCB14] Thanassis Avgerinos, Alexandre Rebert, Sang Kil Cha, and David Brumley. Enhancing symbolic execution with veritesting. In *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*, pages 1083–1094, 2014.

[BBKK12] Sebastian Biallas, Jörg Brauer, Andy King, and Stefan Kowalewski. Loop leaping with closures. In *Static Analysis - 19th International Symposium, SAS 2012, Deauville, France, September 11-13, 2012. Proceedings*, pages 214–230, 2012.

[BCE08] Peter Boonstoppel, Cristian Cadar, and Dawson R. Engler. Rwset: Attacking path explosion in constraint-based test generation. In *Tools and*

Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings, pages 351–366, 2008.

[BGM13] Ella Bounimova, Patrice Godefroid, and David A. Molnar. Billions and billions of constraints: whitebox fuzz testing in production. In *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, pages 122–131, 2013.

[BJAS11] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J. Schwartz. BAP: A binary analysis platform. In *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, pages 463–469, 2011.

[Cad15] Cristian Cadar. Targeted program transformations for symbolic execution. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*, pages 906–909, 2015.

[CARB12] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. Unleashing mayhem on binary code. In *IEEE Symposium on Security and Privacy, SP 2012, 21-23 May 2012, San Francisco, California, USA*, pages 380–394, 2012.

[CDE08] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*, pages 209–224, 2008.

[CF00] Thomas Colcombet and Pascal Fradet. Enforcing trace properties by program transformation. In *POPL 2000, Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Boston, Massachusetts, USA, January 19-21, 2000*, pages 54–66, 2000.

[cgc16] Cyber grand challenge (cgc). https://www.darpa.mil/program/cyber-grand-challenge, 2016.

[CGP+06] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. EXE: automatically generating inputs of death. In *Proceedings of the 13th ACM Conference on Computer and Communications Security, CCS 2006, Alexandria, VA, USA, Ioctober 30 - November 3, 2006*, pages 322–335, 2006.

[CKC12] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. The S2E platform: Design, implementation, and applications. *ACM Trans. Comput. Syst.*, 30(1):2:1–2:49, 2012.

[CKGJ11] Omar Chebaro, Nikolai Kosmatov, Alain Giorgetti, and Jacques Julliand. The SANTE tool: Value analysis, program slicing and test generation for C program debugging. In *Tests and Proofs - 5th International Conference, TAP 2011, Zurich, Switzerland, June 30 - July 1, 2011. Proceedings*, pages 78–83, 2011.

[Cla76] Lori A. Clarke. A system to generate test data and symbolically execute programs. *IEEE Trans. Software Eng.*, 2(3):215–222, 1976.

[CS13] Cristian Cadar and Koushik Sen. Symbolic execution for software testing: three decades later. *Commun. ACM*, 56(2):82–90, 2013.

[DBF+16] Robin David, Sébastien Bardin, Josselin Feist, Laurent Mounier, Marie-Laure Potet, Thanh Dinh Ta, and Jean-Yves Marion. Specification of concretization and symbolization policies in symbolic execution. In *ISSTA, Saarbrücken, Germany, July 18-20, 2016*, pages 36–46, 2016.

[DBG10] Mickaël Delahaye, Bernard Botella, and Arnaud Gotlieb. Explanation-based generalization of infeasible path. In *Third International Conference on Software Testing, Verification and Validation, ICST 2010, Paris, France, April 7-9, 2010*, pages 215–224, 2010.

[DBT+16] Robin David, Sébastien Bardin, Thanh Dinh Ta, Laurent Mounier, Josselin Feist, Marie-Laure Potet, and Jean-Yves Marion. BINSEC/SE: A Dynamic Symbolic Execution Toolkit for Binary-Level Analysis. In *IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering, SANER 2016, Osaka, Japan, March 14-18, 2016 - Volume 1*, pages 653–656, 2016.

[dOBP16] Steven de Oliveira, Saddek Bensalem, and Virgile Prevosto. Polynomial invariants by linear algebra. In *Automated Technology for Verification and Analysis - 14th International Symposium, ATVA 2016, Chiba, Japan, October 17-20, 2016, Proceedings*, pages 479–494, 2016.

[GD07] Vijay Ganesh and David L. Dill. A decision procedure for bit-vectors and arrays. In *Computer Aided Verification, 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007, Proceedings*, pages 519–531, 2007.

[GKS05] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: directed automated random testing. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12-15, 2005*, pages 213–223, 2005.

[GL11] Patrice Godefroid and Daniel Luchaup. Automatic partial loop summarization in dynamic test generation. In *Proceedings of the 20th International Symposium on Software Testing and Analysis, ISSTA 2011, Toronto, ON, Canada, July 17-21, 2011*, pages 23–33, 2011.

[GLM12] Patrice Godefroid, Michael Y. Levin, and David A. Molnar. SAGE: Whitebox Fuzzing for Security Testing. *ACM Queue*, 10(1):20, 2012.

[God07] Patrice Godefroid. Compositional dynamic test generation. In *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2007, Nice, France, January 17-19, 2007*, pages 47–54, 2007.

[Kin76] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.

[KKBC12] Volodymyr Kuznetsov, Johannes Kinder, Stefan Bucur, and George Candea. Efficient state merging in symbolic execution. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*, pages 193–204, 2012.

[NS05] James Newsome and Dawn Xiaodong Song. Dynamic taint analysis for automatic detection, analysis, and signaturegeneration of exploits on commodity software. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2005, San Diego, California, USA*, 2005.

[SAB10] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic

execution (but might have been afraid to ask). In *31st IEEE Symposium on Security and Privacy, S&P 2010, 16-19 May 2010, Berleley/Oakland, California, USA*, pages 317–331, 2010.

[SBY+08] Dawn Xiaodong Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Prateek Saxena. Bitblaze: A new approach to computer security via binary analysis. In *Information Systems Security, 4th International Conference, ICISS 2008, Hyderabad, India, December 16-20, 2008. Proceedings*, pages 1–25, 2008.

[SMA05] Koushik Sen, Darko Marinov, and Gul Agha. CUTE: a concolic unit testing engine for C. In *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2005, Lisbon, Portugal, September 5-9, 2005*, pages 263–272, 2005.

[SSJ+15] Daniel Schwartz-Narbonne, Martin Schäf, Dejan Jovanovic, Philipp Rmmer, and Thomas Wies. Conflict-directed graph coverage. In *NASA Formal Methods - 7th International Symposium, NFM 2015, Pasadena, CA, USA, April 27-29, 2015, Proceedings*, pages 327–342, 2015.

[SST13] Jiri Slaby, Jan Strejcek, and Marek Trtík. Compact symbolic execution. In *Automated Technology for Verification and Analysis - 11th International Symposium, ATVA 2013, Hanoi, Vietnam, October 15-18, 2013. Proceedings*, pages 193–207, 2013.

[Tip95] Frank Tip. A survey of program slicing techniques. *J. Prog. Lang.*, 3(3), 1995.

[Wei81] Mark Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering, San Diego, California, USA, March 9-12, 1981.*, pages 439–449, 1981.

[WMMR05] Nicky Williams, Bruno Marre, Patricia Mouy, and Muriel Roger. Pathcrawler: Automatic generation of path tests by combining static and dynamic analysis. In *Dependable Computing - EDCC-5, 5th European Dependable Computing Conference, Budapest, Hungary, April 20-22, 2005, Proceedings*, pages 281–292, 2005.

Part III

Contributions

Chapter 5

Model Generation for Quantified Formulas: A Taint-Based Approach

We focus in this chapter on generating models for quantified first-order formulas over built-in theories as defined in Chapter 3, which is paramount in software verification and bug finding. Chapter 2 expressed the need of quantified formulas to encode some security properties or to model interactions with the environment. While standard methods are either geared toward proving the absence of a solution rather than finding some, or targeted to specific theories, we propose a correct approach for model generation of quantified formulas modulo theories, based on a reduction to the quantifier-free case through the inference of independence conditions. Our technique is applicable to any theory with a decidable quantifier-free case, and thus allows to reuse all the efficient machinery developed for that context. Experiments show a substantial improvement over state-of-the-art methods. *The content of this chapter was presented in [RL18] and published in [FBBP18].*

5.1 Introduction

Context As stated in Chapter 3, software verification methods have come to rely increasingly on reasoning over logical formulas modulo theory. In particular, the ability to generate models (i.e., find solutions) of a formula is of utmost importance, typically in the context of bug finding or intensive testing — Symbolic Execution [GLM12] or Bounded Model Checking [Bie09]. Since *quantifier-free first-order formulas* on well-suited theories are sufficient to represent many reachability properties of interest, the Satisfiability Modulo Theory (SMT) [BSST09, KS08] community has primarily

dedicated itself to designing solvers able to efficiently handle such problems.

Yet, universal quantifiers are sometimes needed, typically when considering pre-conditions or code abstraction, or when modeling some complex system as shown in Chapter 2. Unfortunately, most theories handled by SMT-solvers are undecidable in the presence of universal quantifiers. There exist dedicated methods for a few decidable quantified theories, such as Presburger arithmetic [BKRW11] or the array property fragment [BMS06], but there is no general and effective enough approach for the model generation problem over universally quantified formulas. Indeed, generic solutions for quantified formulas involving heuristic instantiation and refutation are best geared to prove the unsatisfiability of a formula (i.e., absence of solution) [dMB07, GdM09], while recent proposals such as local theory extensions [BRK+15], finite instantiation [RTGK13, RTG+13] or model-based instantiation [RDK+15, GdM09] either are too narrow in scope, or handle quantifiers on free sorts only, or restrict themselves to finite models, or may get stuck in infinite refinement loops.

Goal and Challenge Our goal is to propose a generic and efficient approach to the model generation problem over arbitrary quantified formulas with support for theories commonly found in software verification. Due to the huge effort made by the community to produce state-of-the-art solvers for quantifier-free theories (*QF-solvers*), it is highly desirable for this solution to be compatible with current leading decision procedures, namely SMT approaches.

Proposal Our approach turns a quantified formula into a quantifier-free formula with the guarantee that any model of the latter contains a model of the former. The benefits are threefold: the transformed formula is easier to solve, it can be sent to standard QF-solvers, and a model for the initial formula is deducible from a model of the transformed one. The idea is to ignore quantifiers but strengthen the quantifier-free part of the formula with an *independence condition* constraining models to be independent from the (initially) quantified variables.

Contributions This chapter makes the following contributions:

- We propose a novel and generic framework for model generation of quantified formula (Section 5.4, Algorithm 5.1) relying on the inference of *sufficient independence condition* (Section 5.3). We prove its *correctness* (Theorem 5.1, mechanized in Coq) and its *efficiency* under reasonable assumptions (Proposition 5.4 and Proposition 5.5). Especially our approach implies only a linear overhead in the

formula size. We also briefly study its *completeness*, related to the notion of *weakest independence condition*.

- We define a taint-based procedure for the inference of independence conditions (Section 5.4.2), composed of a theory-independent core (Algorithm 5.2) together with theory-dependent refinements. We propose such refinements for a large class of operators (Section 5.5.2), encompassing notably arrays and bitvectors.

- Finally, we present a concrete implementation of our method specialized on arrays and bitvectors (Section 5.6). Experiments on SMT-LIB benchmarks and software verification problems notably demonstrate that we are able not only to very effectively lift quantifier-free decision procedures to the quantified case, but also to supplement recent advances, such as finite or model-based quantifier instantiation [RTGK13, RTG+13, RDK+15, GdM09]. Indeed, we concretely supply SMT solvers with the ability to efficiently address an extended set of software verification questions.

Discussions In Section 5.7 we compare our approach to several quantified formulas resolution techniques. Our approach supplements state-of-the-art model generation on quantified formulas by providing a more generic handling of satisfiable problems. We can deal with quantifiers on any sort and we are not restricted to finite models. Moreover, this is a lightweight preprocessing approach requiring a single call to the underlying quantifier-free solver. The method also extends to *partial* elimination of universal quantifiers, or reduction to *quantified-but-decidable* formulas (Section 5.4.4).

While techniques *a la* E-matching allow to lift quantifier-free solvers to the unsatisfiability checking of quantified formulas, this work provides a mechanism to lift them to the satisfiability checking and model generation of quantified formulas, yielding a more symmetric handling of quantified formulas in SMT. This new approach paves the way to future developments such as the definition of more precise inference mechanisms of independence conditions, the identification of interesting subclasses for which inferring weakest independence conditions is feasible, and the combination with other quantifier instantiation techniques.

Convention We consider the framework of many-sorted first-order logic with equality, and we assume standard definitions of sorts, signatures and terms, as presented in Chapter 3, Section 3.1. Letters $a, b, c \ldots$ denote uninterpreted symbols and variables. Letters $x, y, z \ldots$ denote quantified variables. $\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c}$ denote sets of uninterpreted

symbols. $x, y, z \ldots$ denote sets of quantified variables. Finally, $\mathtt{a}, \mathtt{b}, \mathtt{c} \ldots$ denote valuations of associated (sets of) symbols.

In the rest of this chapter, we assume without loss of generality that all formulas are in Skolem normal form. Recall that any formula ϕ in classical logic can be normalized into a formula ψ in Skolem normal form such that any model of ϕ can be lifted into a model of ψ, and vice versa. This strong relation, much closer to formula equivalence than to formula equisatisfiability, ensures that our correctness and completeness results, all along the chapter, hold for arbitrarily quantified formulas.

5.2 Motivation

In this section we illustrate our approach on a running example. Our general procedure is detailed in Section 5.4, together with some theory-dependent refinements in Section 5.5.

Let us take the code sample in Figure 5.1 and suppose we want to reach function `analyze_me`. For this purpose, we need a model (a.k.a., solution) of the reachability condition $\phi \triangleq ax + b > 0$, where a, b and x are symbolic variables associated to the program variables `a`, `b` and `x`. However, while the values of `a` and `b` are user-controlled, the value of `x` is not. Therefore if we want to reach `analyze_me` in a reproducible manner, we actually need a model of $\phi_\forall \triangleq \forall x.ax + b > 0$, which *involves universal quantification*. While this specific formula is simple, model generation for quantified formulas is notoriously difficult: PSPACE-complete for booleans, undecidable for uninterpreted functions or arrays.

Reduction to the Quantifier-Free Case Through Independence We propose to ignore the universal quantification over x, but *restrict models to those which do not depend on x*. For example, model $\{\mathtt{a} = 1, \mathtt{x} = 1, \mathtt{b} = 0\}$ does depend on x, as taking $x = 0$ invalidates the formula, while model $\{\mathtt{a} = 0, \mathtt{x} = 1, \mathtt{b} = 1\}$ is *independent of* x. We call constraint $\psi \triangleq (a = 0)$ an *independence condition*: any interpretation of ϕ satisfying ψ will be independent of x, and therefore a model of $\phi \wedge \psi$ will give us a model of $\phi_\forall$.

Inference of Independence Conditions Through Tainting Figure 5.1 details in its right part a way to infer such independence conditions. Given a quantified reachability condition, here (1) $\forall x.ax + b > 0$, we associate to every variable v a (boolean) *taint variable* $v^\bullet$ indicating whether the solution may depend on v (value $\top$) or not (value $\bot$). Here (2), $x^\bullet$ is set to $\bot$, $a^\bullet$ and $b^\bullet$ are set to $\top$. An independence condition (3) — a formula modulo theory — is then constructed using both initial and taint variables.

```
int main () {
  int a = input ();
  int b = input ();

  int x = rand ();

  if (a * x + b > 0) {
    analyze_me();
  }
  else {
    ...;
  }
}
```

Quantified reachability condition

(1) $\forall x.ax + b > 0$

Taint variable constraint

(2) $a^\bullet \wedge b^\bullet \wedge \neg(x^\bullet)$ $(a^\bullet, b^\bullet, x^\bullet$: fresh boolean)

Independence condition

(3) $((a^\bullet \wedge x^\bullet) \vee (a^\bullet \wedge a = 0) \vee (x^\bullet \wedge x = 0)) \wedge b^\bullet$

(4) $((\top \wedge \bot) \vee (\top \wedge a = 0) \vee (\bot \wedge x = 0)) \wedge \top$

(5) $a = 0$

Quantifier-free approximation of (1)

(6) $(ax + b > 0) \wedge (a = 0)$

Figure 5.1 – Motivating example.

Condition (3) comes from the fact that in order to enforce $ax + b > 0$ to be independent from x, we have to enforce that ax and b are independent from x. b is independent from x if $b^\bullet$ hold, while ax is independent from x if: either $a^\bullet$ and $x^\bullet$ hold; or $a^\bullet$ holds and $a = 0$ (absorbing the value of x); or the symmetric case. We see that taint constraints are defined recursively and combines a *systematic part* (if t is independent from x then $f(t)$ also is, for any f) with a *theory-dependent part* (here, based on the absorption property of $\times$). After simplifications (4), we obtain $a = 0$ as an independence condition (5) which is adjoined to the reachability condition freed of its universal quantification (6). A QF-solver provides a model of (6) (e.g., $\{\mathtt{a} = 0, \mathtt{b} = 1, \mathtt{x} = 5\}$), lifted into a model of (1) by discarding the valuation of x (e.g., $\{\mathtt{a} = 0, \mathtt{b} = 1\}$).

In this specific example the inferred independence condition (5) is the most generic one and (1) and (6) are equisatisfiable. Yet, in general it may be an under-approximation, constraining the variables more than needed and yielding a correct but incomplete decision method: a model of (6) can still be turned into a model of (1), but (6) might not have a model while (1) has.

5.3 Musing with Independence

In this section we introduce the notion of independence, the formal ground on which our model generation approach stands. We first define independence for interpretations, terms and formulas. Then we explain what are independence conditions and how they

relate to model generation.

5.3.1 Independent Interpretations, Terms and Formulas

A solution $(\mathtt{x}, \mathtt{a})$ of Φ does not depend on $\boldsymbol{x}$ if $\Phi(\boldsymbol{x}, \boldsymbol{a})$ is always true or always false, for all possible valuations of $\boldsymbol{x}$ as long as $\boldsymbol{a}$ is set to $\mathtt{a}$. More formally, we define the independence of an interpretation of Φ with regard to $\boldsymbol{x}$ as follows:

Definition 5.1 (Independent Interpretation)**.**

- Let $\Phi(\boldsymbol{x}, \boldsymbol{a})$ a formula with free variables $\boldsymbol{x}$ and $\boldsymbol{a}$. Then an interpretation $\mathcal{I}$ of $\Phi(\boldsymbol{x}, \boldsymbol{a})$ is independent of $\boldsymbol{x}$ if for all interpretations $\mathcal{J}$ equal to $\mathcal{I}$ except on $\boldsymbol{x}$, $\mathcal{I} \models \Phi$ if and only if $\mathcal{J} \models \Phi$.

- Let $\Delta(\boldsymbol{x}, \boldsymbol{a})$ a term with free variables $\boldsymbol{x}$ and $\boldsymbol{a}$. Then an interpretation $\mathcal{I}$ of $\Delta(\boldsymbol{x}, \boldsymbol{a})$ is independent of $\boldsymbol{x}$ if for all interpretations $\mathcal{J}$ equal to $\mathcal{I}$ except on $\boldsymbol{x}$, $[\![\Delta(\boldsymbol{x}, \boldsymbol{a})]\!]_{\mathcal{I}} = [\![\Delta(\boldsymbol{x}, \boldsymbol{a})]\!]_{\mathcal{J}}$.

Regarding formula $ax + b > 0$ from Figure 5.1, $\{a = 0, b = 1, x = 1\}$ is independent of x while $\{a = 1, b = 0, x = 1\}$ is not. Considering term $(t\,[a \leftarrow b])\,[c]$, with t an array written at index a then read at index c, $\{a = 0, b = 42, c = 0, t = [\ldots]\}$ is independent of t (evaluates to 42) while $\{a = 0, b = 1, c = 2, t = [\ldots]\}$ is not (evaluates to $t\,[2]$). We now define independence for formulas and terms.

Definition 5.2 (Independent Formula and Term)**.**

- Let $\Phi(\boldsymbol{x}, \boldsymbol{a})$ a formula with free variables $\boldsymbol{x}$ and $\boldsymbol{a}$. Then $\Phi(\boldsymbol{x}, \boldsymbol{a})$ is independent of $\boldsymbol{x}$ if $\forall \boldsymbol{x}.\forall \boldsymbol{y}.\ (\Phi(\boldsymbol{x}, \boldsymbol{a}) \Leftrightarrow \Phi(\boldsymbol{y}, \boldsymbol{a}))$ is true for any value of $\boldsymbol{a}$.

- Let $\Delta(\boldsymbol{x}, \boldsymbol{a})$ a term with free variables $\boldsymbol{x}$ and $\boldsymbol{a}$. Then $\Delta(\boldsymbol{x}, \boldsymbol{a})$ is independent of $\boldsymbol{x}$ if $\forall \boldsymbol{x}.\forall \boldsymbol{y}.\ (\Delta(\boldsymbol{x}, \boldsymbol{a}) = \Delta(\boldsymbol{y}, \boldsymbol{a}))$ is true for any value of $\boldsymbol{a}$.

Definition 5.2 of formula and term independence is far stronger than Definition 5.1 of interpretation independence. Indeed, it can easily be checked that if a formula Φ (resp. a term Δ) is independent of $\boldsymbol{x}$, then any interpretation of Φ (resp. Δ) is independent of $\boldsymbol{x}$. However, the converse is false as formula $ax + b > 0$ is not independent of x, but has an interpretation $\{a = 0, b = 1, x = 1\}$ which is.

5.3.2 Independence Conditions

Since it is rarely the case that a formula (resp. term) is independent from a set of variables $\boldsymbol{x}$, we are interested in *Sufficient Independence Conditions*. These conditions are additional constraints that can be added to a formula (resp. term) in such a way that they make the formula (resp. term) independent of $\boldsymbol{x}$.

Definition 5.3 (Sufficient Independence Condition (SIC))**.**

- A Sufficient Independence Condition for a formula $\Phi(\boldsymbol{x}, \boldsymbol{a})$ with regard to $\boldsymbol{x}$ is a formula $\Psi(\boldsymbol{a})$ such that $\Psi(\boldsymbol{a}) \models (\forall \boldsymbol{x}.\forall \boldsymbol{y}.\Phi(\boldsymbol{x}, \boldsymbol{a}) \Leftrightarrow \Phi(\boldsymbol{y}, \boldsymbol{a}))$.
- A Sufficient Independence Condition for a term $\Delta(\boldsymbol{x}, \boldsymbol{a})$ with regard to $\boldsymbol{x}$, is a formula $\Psi(\boldsymbol{a})$ such that $\Psi(\boldsymbol{a}) \models (\forall \boldsymbol{x}.\forall \boldsymbol{y}.\Delta(\boldsymbol{x}, \boldsymbol{a}) = \Delta(\boldsymbol{y}, \boldsymbol{a}))$.

We denote by $\text{SIC}_{\Phi,\boldsymbol{x}}$ (resp. $\text{SIC}_{\Delta,\boldsymbol{x}}$) a Sufficient Independence Condition for a formula $\Phi(\boldsymbol{x}, \boldsymbol{a})$ (resp. for a term $\Delta(\boldsymbol{x}, \boldsymbol{a})$) with regard to $\boldsymbol{x}$. For example, $a = 0$ is a $\text{SIC}_{\Phi,x}$ for formula $\Phi \triangleq ax + b > 0$, and $a = c$ is a $\text{SIC}_{\Delta,t}$ for term $\Delta \triangleq (t\,[a \leftarrow b])\,[c]$. Note that $\bot$ is always a SIC, and that SIC are closed under $\wedge$ and $\vee$. Proposition 5.1 clarifies the interest of SIC for model generation.

Proposition 5.1 (Model Generalization)**.** *Let $\Phi(\boldsymbol{x}, \boldsymbol{a})$ a formula and Ψ a $\text{SIC}_{\Phi,\boldsymbol{x}}$. If there exists an interpretation $\{\mathsf{x}, \mathsf{a}\}$ such that $\{\mathsf{x}, \mathsf{a}\} \models \Psi(\boldsymbol{a}) \wedge \Phi(\boldsymbol{x}, \boldsymbol{a})$, then $\{\mathsf{a}\} \models \forall \boldsymbol{x}.\Phi(\boldsymbol{x}, \boldsymbol{a})$.*

Proof. Let (x, a) an interpretation of $\Phi(\boldsymbol{x}, \boldsymbol{a})$, and let us assume that $(\mathsf{x}, \mathsf{a}) \models \Psi(\boldsymbol{a}) \wedge \Phi(\boldsymbol{x}, \boldsymbol{a})$. It comes immediately that $(\mathsf{x}, \mathsf{a}) \models \Psi(\boldsymbol{a})$ (E1) and $(\mathsf{x}, \mathsf{a}) \models \Phi(\boldsymbol{x}, \boldsymbol{a})$ (E2). From (E1) we deduce that $\mathsf{a} \models \Psi(\boldsymbol{a})$, since $\boldsymbol{x}$ does not appear in Ψ. Then, Ψ being a $\text{SIC}_{\Phi,\boldsymbol{x}}$, we get that $\forall \boldsymbol{x}.\forall \boldsymbol{y}.\ (\Phi(\boldsymbol{x}, \mathsf{a}) \Leftrightarrow \Phi(\boldsymbol{y}, \mathsf{a}))$ is true (E3). Combining (E3) with the fact that $\Phi(\mathsf{x}, \mathsf{a})$ is true (from E2), we conclude that $\Phi(\boldsymbol{x}, \mathsf{a})$ is satisfied for any value of $\boldsymbol{x}$, hence $\mathsf{a} \models \forall \boldsymbol{x}.\Phi(\boldsymbol{x}, \boldsymbol{a})$. □

For the sake of completeness, we introduce now the notion of *Weakest Independence Condition* for a formula $\Phi(\boldsymbol{x}, \boldsymbol{a})$ with regard to $\boldsymbol{x}$ (resp. a term $\Delta(\boldsymbol{x}, \boldsymbol{a})$). We will denote such conditions $\text{WIC}_{\Phi,\boldsymbol{x}}$ (resp. $\text{WIC}_{\Delta,\boldsymbol{x}}$).

Definition 5.4 (Weakest Independence Condition (WIC))**.**

- A Weakest Independence Condition for a formula $\Phi(\boldsymbol{x}, \boldsymbol{a})$ with regard to $\boldsymbol{x}$ is a $\text{SIC}_{\Phi,\boldsymbol{x}}$ Π such that, for any other $\text{SIC}_{\Phi,\boldsymbol{x}}$ Ψ, $\Psi \models \Pi$.
- A Weakest Independence Condition for a term $\Delta(\boldsymbol{x}, \boldsymbol{a})$ with regard to $\boldsymbol{x}$ is a $\text{SIC}_{\Delta,\boldsymbol{x}}$ Π such that, for any other $\text{SIC}_{\Delta,\boldsymbol{x}}$ Ψ, $\Psi \models \Pi$.

Note that $\Omega \triangleq \forall \boldsymbol{x}.\forall \boldsymbol{y}.\,(\Phi\,(\boldsymbol{x},\boldsymbol{a}) \Leftrightarrow \Phi\,(\boldsymbol{y},\boldsymbol{a}))$ is always a $\text{WIC}_{\Phi,\boldsymbol{x}}$, and any formula Π is a $\text{WIC}_{\Phi,\boldsymbol{x}}$ if and only if $\Pi \equiv \Omega$. Therefore all syntactically different WIC have the same semantics. As an example, both SIC $a = 0$ and $a = c$ presented earlier are WIC. Proposition 5.2 emphasizes the interest of WIC for model generation.

Proposition 5.2 (Model Specialization)**.** *Let $\Phi\,(\boldsymbol{x},\boldsymbol{a})$ a formula and $\Pi(\boldsymbol{a})$ a $\text{WIC}_{\Phi,\boldsymbol{x}}$. If there exists an interpretation $\{\mathsf{a}\}$ such that $\{\mathsf{a}\} \models \forall \boldsymbol{x}.\Phi\,(\boldsymbol{x},\boldsymbol{a})$, then $\{\boldsymbol{x},\mathsf{a}\} \models \Pi\,(\boldsymbol{a}) \wedge \Phi\,(\boldsymbol{x},\boldsymbol{a})$ for any valuation $\mathbf{x}$ of $\boldsymbol{x}$.*

Proof. Let a an interpretation such that $\{\mathsf{a}\} \models \forall \boldsymbol{x}.\Phi\,(\boldsymbol{x},\boldsymbol{a})$ (E1). Then by definition, $\forall \boldsymbol{x}.\Phi\,(\boldsymbol{x},\mathsf{a})$ is true. Especially $\forall \boldsymbol{x}.\forall \boldsymbol{y}.\,(\Phi\,(\boldsymbol{x},\mathsf{a}) \Leftrightarrow \Phi\,(\boldsymbol{x},\mathsf{a}))$ also is, and therefore $\{\mathsf{a}\} \models \Pi(\boldsymbol{a})$ (E2). Let us now consider an arbitrary value $\mathbf{x}$ for $\boldsymbol{x}$. By combining (E2) and (E1), we obtain that $(\mathbf{x},\mathsf{a}) \models \Pi\,(\boldsymbol{a}) \wedge \Phi\,(\boldsymbol{x},\boldsymbol{a})$. □

From now on, our goal is to infer from a formula $\forall \boldsymbol{x}.\Phi\,(\boldsymbol{x},\boldsymbol{a})$ a $\text{SIC}_{\Phi,\boldsymbol{x}}$ $\Psi\,(\boldsymbol{a})$, find a model for $\Psi\,(\boldsymbol{a}) \wedge \Phi\,(\boldsymbol{x},\boldsymbol{a})$ and generalize it. This $\text{SIC}_{\Phi,\boldsymbol{x}}$ should be as weak — in the sense "less coercive" — as possible, as otherwise $\bot$ could always be used, which would not be very interesting for our overall purpose.

For the sake of simplicity, previous definitions omit to mention the theory to which the SIC belongs. If the theory $\mathcal{T}$ of the quantified formula is decidable we can always choose $\forall \boldsymbol{x}.\forall \boldsymbol{y}.\,(\Phi\,(\boldsymbol{x},\boldsymbol{a}) \Leftrightarrow \Phi\,(\boldsymbol{y},\boldsymbol{a}))$ as a SIC, but it is simpler to directly use a $\mathcal{T}$-solver. *The challenge is, for formulas in an undecidable theory $\mathcal{T}$, to find a non-trivial SIC in its quantifier-free fragment QF-$\mathcal{T}$.*

Under this constraint, we cannot expect a systematic construction of WIC, as it would allow to decide the satisfiability of any quantified theory with a decidable quantifier-free fragment. Yet informally, the closer a SIC is to be a WIC, the closer our approach is to completeness. Therefore this notion might be seen as a fair gauge of the quality of a SIC. *Having said that, we leave a deeper study on the inference of WIC as future work.*

5.4 Generic Framework for SIC-Based Model Generation

We describe now our overall approach. Algorithm 5.1 presents our SIC-based generic framework for model generation (Section 5.4.1). Then, Algorithm 5.2 proposes a taint-based approach for SIC inference (Section 5.4.2). Finally, we discuss complexity and efficiency issues (Section 5.4.3) and detail extensions (Section 5.4.4), such as partial elimination.

From now on, we do not distinguish anymore between terms and formulas, their treatment being symmetric, and we call targeted variables the variables we want to be independent of.

5.4.1 SIC-Based Model Generation

Algorithm 5.1: SIC-based model generation for quantified formulas.

Parameter: `solveQF`
- **Input:** $\Phi(v)$ a formula in QF-$\mathcal{T}$
- **Output:** SAT $(\mathtt{v})$ with $\mathtt{v} \models \Phi$, UNSAT or UNKNOWN

Parameter: `inferSIC`
- **Input:** Φ a formula in QF-$\mathcal{T}$, and $\boldsymbol{x}$ a set of targeted variables
- **Output:** Ψ a $\text{SIC}_{\Phi,\boldsymbol{x}}$ in QF-$\mathcal{T}$

Function `solveQ`:
- **Input:** $\forall \boldsymbol{x}.\Phi(\boldsymbol{x}, \boldsymbol{a})$ a universally quantified formula over theory $\mathcal{T}$
- **Output:** SAT $(\mathtt{a})$ with $\mathtt{a} \models \forall \boldsymbol{x}.\Phi(\boldsymbol{x}, \boldsymbol{a})$, UNSAT or UNKNOWN
- Let $\Psi(\boldsymbol{a}) \triangleq$ `inferSIC` $(\Phi(\boldsymbol{x}, \boldsymbol{a}), \boldsymbol{x})$
- **match** `solveQF` $(\Phi(\boldsymbol{x}, \boldsymbol{a}) \wedge \Psi(\boldsymbol{a}))$
  - **with** *SAT* $(\boldsymbol{x}, \mathtt{a})$ **return** *SAT* $(\mathtt{a})$
  - **with** *UNSAT*
    - **if** Ψ *is a* $\text{WIC}_{\Phi,\boldsymbol{x}}$ **then return** *UNSAT*
    - **else return** *UNKNOWN*
  - **with** *UNKNOWN* **return** *UNKNOWN*

Our model generation technique is described in Algorithm 5.1. Function `solveQ` takes as input a formula $\forall \boldsymbol{x}.\Phi(\boldsymbol{x}, \boldsymbol{a})$ over a theory $\mathcal{T}$. It first calculates a $\text{SIC}_{\Phi,\boldsymbol{x}}$ $\Psi(\boldsymbol{a})$ in QF-$\mathcal{T}$. Then it solves $\Phi(\boldsymbol{x}, \boldsymbol{a}) \wedge \Psi(\boldsymbol{a})$. Finally, depending on the result and whether $\Psi(\boldsymbol{a})$ is a $\text{WIC}_{\Phi,\boldsymbol{x}}$ or not, it answers SAT, UNSAT or UNKNOWN. `solveQ` is parametrized by two functions `solveQF` and `inferSIC`:

`solveQF` is a decision procedure (typically a SMT solver) for QF-$\mathcal{T}$. `solveQF` is said to be *correct* if each time it answers SAT (resp. UNSAT) the formula is satisfiable (resp. unsatisfiable); it is said to be *complete* if it always answers SAT or UNSAT, never UNKNOWN.

`inferSIC` takes as input a formula Φ in QF-$\mathcal{T}$ and a set of targeted variables $\boldsymbol{x}$, and produces a $\text{SIC}_{\Phi,\boldsymbol{x}}$ in QF-$\mathcal{T}$. It is said to be *correct* if it always returns a SIC, and *complete* if all the SIC it returns are WIC. A possible implementation of `inferSIC` is described in Algorithm 5.2 (Section 5.4.2).

Function `solveQ` enjoys the two following properties, where correctness and completeness are defined as for `solveQF`.

Theorem 5.1 (Correctness and Completeness)**.**

- *If* `solveQF` *and* `inferSIC` *are correct, then* `solveQ` *is correct.*
- *If* `solveQF` *and* `inferSIC` *are complete, then* `solveQ` *is complete.*

Proof. Follow directly from Proposition 5.1 and Proposition 5.2, Section 5.3.2. □

5.4.2 Taint-Based SIC Inference

Algorithm 5.2: Taint-based SIC inference.

Parameter: `theorySIC`

Input: f a function symbol, its parameters ϕ_i, $\boldsymbol{x}$ a set of targeted variables and ψ_i their associated $\text{SIC}_{\phi_i,\boldsymbol{x}}$

Output: Ψ a $\text{SIC}_{f(\phi_i),\boldsymbol{x}}$

Default: Return $\bot$

Function `inferSIC`$(\Phi,\boldsymbol{x})$**:**

Input: Φ a formula and $\boldsymbol{x}$ a set of targeted variables

Output: Ψ a $\text{SIC}_{\Phi,\boldsymbol{x}}$

either Φ *is a constant* **return** $\top$

either Φ *is a variable* v **return** $v \notin \boldsymbol{x}$

either Φ *is a function* $f\ (\phi_1,..,\phi_n)$

Let $\psi_i \triangleq$ `inferSIC` $(\phi_i, \boldsymbol{x})$ for all $i \in \{1,..,n\}$

Let $\Psi \triangleq$ `theorySIC` $(f, (\phi_1,..,\phi_n), (\psi_1,..,\psi_n), \boldsymbol{x})$

return $\Psi \vee \bigwedge_i \psi_i$

Algorithm 5.2 presents a taint-based implementation of function `inferSIC`. It consists of a (syntactic) core calculus described here, refined by a (semantic) theory-dependent calculus `theorySIC` described in Section 5.5. From formula $\Phi(\boldsymbol{x}, \boldsymbol{a})$ and targeted variables $\boldsymbol{x}$, `inferSIC` is defined recursively as follows.

If Φ is a constant it returns $\top$ as constants are independent of any variable. If Φ is a variable v, it returns $\top$ if we may depend on v (i.e., $v \notin \boldsymbol{x}$), $\bot$ otherwise. If Φ is a function $f\ (\phi_1,..,\phi_n)$, it first recursively computes for every sub-term ϕ_i a $\text{SIC}_{\phi_i,\boldsymbol{x}}$ ψ_i. Then these results are sent with Φ to `theorySIC` which computes a $\text{SIC}_{\Phi,\boldsymbol{x}}$ Ψ. The

procedure returns the disjunction between Ψ and the conjunction of the ψ_i's. Note that `theorySIC` default value $\bot$ is absorbed by the disjunction.

The intuition is that if the ϕ_i's are independent of $\boldsymbol{x}$, then $f\ (\phi_1, .., \phi_n)$ is. Therefore Algorithm 5.2 is said to be *taint-based* as, when `theorySIC` is left to its default value, it acts as a form of taint tracking [DD77, Ørb95] inside the formula.

Proposition 5.3 (Correctness)**.** *Given a formula $\Phi\ (\boldsymbol{x}, \boldsymbol{a})$ and assuming that* `theorySIC` *is correct, then* `inferSIC` $(\Phi, \boldsymbol{x})$ *indeed computes a* $\text{SIC}_{\Phi,\boldsymbol{x}}$.

Proof. This proof has been mechanized in Coq[1]. □

Note that on the other hand, completeness does not hold: in general `inferSIC` does not compute a WIC, cf. discussion in Section 5.4.4.

5.4.3 Complexity and Efficiency

We now evaluate the overhead induced by Algorithm 5.1 in terms of formula size and complexity of the resolution — the running time of Algorithm 5.1 itself being expected to be negligible (preprocessing).

Definition 5.5 (Term Size)**.** The size of a term is inductively defined as $\text{size}\ (x) \triangleq 1$ for x a variable, and $\text{size}\ (f\ (t_1, .., t_n)) \triangleq 1 + \Sigma_i\ \text{size}\ (t_i)$ otherwise. We say that `theorySIC` is bounded in size if there exists K such that, for all terms Δ, $\text{size}\ (\texttt{theorySIC}\ (\Delta, \cdot)) \leq K$.

Proposition 5.4 (Size Bound)**.** *Let N be the maximal arity of symbols defined by theory $\mathcal{T}$. If* `theorySIC` *is bounded in size by K, then for all formula Φ in $\mathcal{T}$, size* $(\texttt{inferSIC}\ (\Phi, \cdot)) \leq (K + N) \cdot \textit{size}\ (\Phi)$.

Proof. Let $\Phi \triangleq f\ (\phi_1, .., \phi_n)$ be a formula, let $\psi_i \triangleq \texttt{inferSIC}\ (\phi_i)$ be results of recursive calls, and let $\Psi \triangleq \texttt{theorySIC}\ (f, (\phi_1, .., \phi_n), (\psi_1, .., \psi_n), \boldsymbol{x})$.

$$\begin{aligned} \text{size}\ (\texttt{inferSIC}\ (f\ (\phi_1, \ldots, \phi_n))) &= \text{size}\ (\Psi \vee \textstyle\bigwedge_i \psi_i) \\ &= \text{size}\ (\Psi) + 1 + (n-1) + \Sigma_i \text{size}\ (\psi_i) \\ &\leq \quad K \quad + \quad N \quad + \Sigma_i \text{size}\ (\psi_i) \end{aligned}$$

Then by structural induction we have $\text{size}\ (\texttt{inferSIC}\ (\Phi, \cdot)) \leq (K + N) \cdot \text{size}\ (\Phi)$. □

Proposition 5.5 (Complexity Bound)**.** *Let us suppose* `theorySIC` *bounded in size, and let Φ be a formula belonging to a theory $\mathcal{T}$ with polynomial-time checkable solutions. If Ψ is a* $\text{SIC}_{\Phi,\cdot}$ *produced by* `inferSIC`*, then a solution for $\Phi \wedge \Psi$ is checkable in time polynomial in size of Φ.*

[1] http://benjamin.farinier.org/cav2018/

Proof. If `theorySIC` is bounded in size, then `inferSIC` is linear in size by Proposition 5.4. So for any formula Φ in a theory $\mathcal{T}$, if Ψ is the SIC produced by `inferSIC`, then $\Phi \wedge \Psi$ is linearly proportional to Φ. As Ψ lands in a sub-theory of $\mathcal{T}$, $\Phi \wedge \Psi$ belongs to $\mathcal{T}$ and therefore is checkable in polynomial time with regard to the size of Φ. □

These propositions demonstrate that, for formula landing in complex enough theories, our method lifts QF-solvers to the quantified case (in an approximated way) without any significant overhead, as long as `theorySIC` is bounded in size. This latter constraint can be achieved by systematically *let-binding* sub-terms to (constant-size) fresh names and having `theorySIC` manipulates these binders.

5.4.4 Discussions

Extension Let us remark that our framework encompasses partial quantifier elimination as long as the remaining quantifiers are handled by `solveQF`. For example, we may want to remove quantifications over arrays but keep those on bitvectors. In this setting, `inferSIC` can also allow some level of quantification, providing that `solveQF` handles them.

About WIC As already stated, `inferSIC` does not propagate WIC in general. For example, considering formulas $t_1 \triangleq (x < 0)$ and $t_2 \triangleq (x \geq 0)$, then $\text{WIC}_{t_1,x} = \bot$ and $\text{WIC}_{t_2,x} = \bot$. Hence `inferSIC` returns $\bot$ as SIC for $t_1 \vee t_2$, while actually $\text{WIC}_{t_1 \vee t_2,x} = \top$.

Nevertheless, we can already highlight a few cases where WIC can be computed. (1) `inferSIC` does propagate WIC on one-to-one uninterpreted functions. (2) If no variable of $\boldsymbol{x}$ appears in any sub-term of $f(t, t')$, then the associated WIC is $\top$. While a priori naive, this case becomes interesting when combined with simplifications (Section 5.6.1) that may eliminate $\boldsymbol{x}$. (3) If a sub-term falls in a sub-theory admitting quantifier elimination, then the associated WIC is computed by eliminating quantifiers in $(\forall \boldsymbol{x}.\boldsymbol{y}.\Phi(\boldsymbol{x}, \boldsymbol{a}) \Leftrightarrow \Phi(\boldsymbol{y}, \boldsymbol{a}))$. (4) We may also think of dedicated patterns: regarding bitvectors, the WIC for $x \leq a \Rightarrow x \leq x + k$ is $a \leq \texttt{Max} - k$. *Identifying under which condition WIC propagation holds is a strong direction for future work.*

5.5 Theory-Dependent SIC Refinements

We now present theory-dependent SIC refinements for theories relevant to program analysis: booleans, fixed-size bitvectors and arrays — recall that uninterpreted functions

are already handled by Algorithm 5.2. We then propose a generalization of these refinements together with a correctness proof for a larger class of operators.

5.5.1 Refinement on Theories

We recall `theorySIC` takes four parameters: a function symbol f, its arguments $(t_1, .., t_n)$, their associated SIC $\left(t_1^\bullet, .., t_n^\bullet\right)$, and targeted variables $\boldsymbol{x}$. `theorySIC` pattern-matches the function symbol and returns the associated SIC according to rules in Figure 5.2. If a function symbol is not supported, we return the default value $\bot$. Constants and variables are handled by `inferSIC`. For the sake of simplicity, rules in Figure 5.2 are defined recursively, but can easily fit the interface required for `theorySIC` in Algorithm 5.2 by turning recursive calls into parameters.

$$
\begin{aligned}
(a \Rightarrow b)^\bullet &\triangleq (a^\bullet \wedge a = \bot) \vee (b^\bullet \wedge b = \top) \\
(a \wedge b)^\bullet &\triangleq (a^\bullet \wedge a = \bot) \vee (b^\bullet \wedge b = \bot) \\
(a \vee b)^\bullet &\triangleq (a^\bullet \wedge a = \top) \vee (b^\bullet \wedge b = \top) \\
(\textbf{if } c \textbf{ then } a \textbf{ else } b)^\bullet &\triangleq (c^\bullet \wedge \textbf{if } c \textbf{ then } a^\bullet \textbf{ else } b^\bullet) \vee (a^\bullet \wedge b^\bullet \wedge a = b)
\end{aligned}
$$

(a) Booleans and **if** · **then** · **else** ·

$$
\begin{aligned}
(a_n \wedge b_n)^\bullet &\triangleq (a_n^\bullet \wedge a_n = 0_n) \vee (b_n^\bullet \wedge b_n = 0_n) \\
(a_n \vee b_n)^\bullet &\triangleq (a_n^\bullet \wedge a_n = 1_n) \vee (b_n^\bullet \wedge b_n = 1_n) \\
(a_n \times b_n)^\bullet &\triangleq (a_n^\bullet \wedge a_n = 0_n) \vee (b_n^\bullet \wedge b_n = 0_n) \\
(a_n \ll b_n)^\bullet &\triangleq (b_n^\bullet \wedge b_n \geq n)
\end{aligned}
$$

(b) Fixed-size bitvectors

$$
\begin{aligned}
((a[i] \leftarrow e)[j])^\bullet &\triangleq (\textbf{if } (i = j) \textbf{ then } e \textbf{ else } (a[j]))^\bullet \\
&\triangleq ((i = j)^\bullet \wedge (\textbf{if } (i = j) \textbf{ then } e^\bullet \textbf{ else } (a[j])^\bullet)) \\
&\quad \vee (e^\bullet \wedge (a[j])^\bullet \wedge (e = a[j])) \\
&\triangleq (i^\bullet \wedge j^\bullet \wedge (\textbf{if } (i = j) \textbf{ then } e^\bullet \textbf{ else } (a[j])^\bullet)) \\
&\quad \vee (e^\bullet \wedge (a[j])^\bullet \wedge (e = a[j]))
\end{aligned}
$$

(c) Arrays

Figure 5.2 – Examples of refinements for `theorySIC`.

Booleans and if · then · else · Rules for the boolean theory (Figure 5.2a) handles $\Rightarrow$, $\wedge$, $\vee$ and **if** · **then** · **else** ·.For binary operators, the SIC is the conjunction of the SIC associated to one of the two sub-terms and a constraint on this sub-term that forces the

result of the operator to be constant — e.g., to be equal to $\bot$ (resp. $\top$) for the antecedent (resp. consequent) of an implication. These equality constraints are based on absorbing elements of operators.

Inference for the **if** $\cdot$ **then** $\cdot$ **else** $\cdot$ operator is more subtle. Intuitively, if its condition is independent of some x, we use it to select the SIC_x of the sub-term that will be selected by the **if** $\cdot$ **then** $\cdot$ **else** $\cdot$ operator. If the condition is dependent of x, then we cannot use it anymore to select a SIC_x. In this case, we return the conjunction of the SIC_x of both sub-terms and the constraint that the two sub-terms are equal.

Bitvectors and Arrays Rules for bitvectors (Figure 5.2b) follow similar ideas, with constant $\top$ (resp. $\bot$) substituted by 1_n (resp. 0_n), the bitvector of size n full of ones (resp. zeros). Rules for arrays (Figure 5.2c) are derived from the theory axioms. The definition is recursive: rules need be applied until reaching either a $\cdot[\cdot] \leftarrow \cdot$ at the position where the $\cdot[\cdot]$ occurs, or the initial array variable.

As a rule of thumb, good SIC can be derived from function axioms in the form of rewriting rules, as done for arrays. Similar constructions can be obtained for example for stacks or queues.

5.5.2 $\mathcal{R}$-Absorbing Functions

We propose a generalization of the previous theory-dependent SIC refinements to a larger class of functions, and prove its correctness.

Intuitively, if a function has an absorbing element, constraining one of its operands to be equal to this element will ensure that the result of the function is independent of the other operands. However, it is not enough when a relation between some elements is needed, such as with $(t[a \leftarrow b])\,[c]$ where constraint $a = c$ ensures the independence with regard to t. We thus generalize the notion of absorption to $\mathcal{R}$-absorption, where $\mathcal{R}$ is a relation between function arguments.

Definition 5.6 ($\mathcal{R}$-Absorbing Function)**.** Let $f : \tau_1 \times \cdots \times \tau_n \to \tau$ a function. f is $\mathcal{R}$-absorbing if there exists $\mathcal{I}_\mathcal{R} \subset \{1, \cdots, n\}$ and $\mathcal{R}$ a relation between $\alpha_i : \tau_i$, $i \in \mathcal{I}_\mathcal{R}$ such that, for all $b \triangleq (b_1, \ldots, b_n)$ and $c \triangleq (c_1, \ldots, c_n) \in \tau_1 \times \cdots \times \tau_n$, if $\mathcal{R}(b|_{\mathcal{I}_\mathcal{R}})$ and $b|_{\mathcal{I}_\mathcal{R}} = c|_{\mathcal{I}_\mathcal{R}}$ where $\cdot|_{\mathcal{I}_\mathcal{R}}$ is the projection on $\mathcal{I}_\mathcal{R}$, then $f(b) = f(c)$.

$\mathcal{I}_\mathcal{R}$ is called the support of the relation of absorption $\mathcal{R}$.

For example, $(a, b) \mapsto a \vee b$ has two pairs $\langle \mathcal{R}, \mathcal{I}_\mathcal{R} \rangle$ coinciding with the usual notion of absorption, $\langle a = \top, \{1_a\} \rangle$ and $\langle b = \top, \{2_b\} \rangle$. Function $(x, y, z) \mapsto xy + z$ has

$$a \Rightarrow b : \langle a = \bot,\ \{1_a\}\rangle, \langle b = \top,\ \{2_b\}\rangle$$
$$a \wedge b : \langle a = \bot,\ \{1_a\}\rangle, \langle b = \bot,\ \{2_b\}\rangle$$
$$a \vee b : \langle a = \top,\ \{1_a\}\rangle, \langle b = \top,\ \{2_b\}\rangle$$

(a) Booleans

$$a_n \vee b_n : \langle a_n = 1_n,\ \{1_a\}\rangle, \langle b_n = 1_n,\ \{2_b\}\rangle$$
$$a_n \wedge b_n : \langle a_n = 0_n,\ \{1_a\}\rangle, \langle b_n = 0_n,\ \{2_b\}\rangle$$
$$a_n \times b_n : \langle a_n = 0_n,\ \{1_a\}\rangle, \langle b_n = 0_n,\ \{2_b\}\rangle$$

(b) Fixed-size bitvectors

$$\textbf{if}\ \ \textbf{then}\ c\ \textbf{else}\ a\ b : \langle c = \top,\ \{1_c, 2_a\}\rangle, \langle c = \bot,\ \{1_c, 3_b\}\rangle, \langle a = b,\ \{2_a, 3_b\}\rangle$$

(c) if then else

Figure 5.3 – Relation of absorption for some ABV function symbols.

among others the pair $\langle x = 0,\ \{1_x, 3_z\}\rangle$, while $(a, b, c, t) \mapsto (t[a \leftarrow b])\,[c]$ has the pair $\langle a = c,\ \{1_a, 3_c\}\rangle$. We can now state the following proposition:

Proposition 5.6 ($\mathcal{R}$-Absorbing Function are SIC)**.** *Let $f\,(t_1, \ldots, t_n)$ be a $\mathcal{R}$-absorbing function of support $\mathcal{I}_\mathcal{R}$, and let $t_i^\bullet$ be a $\text{SIC}_{t_i,\boldsymbol{x}}$ for some $\boldsymbol{x}$. Then $\mathcal{R}\left(t_{i\in\mathcal{I}_\mathcal{R}}\right) \bigwedge_{i\in\mathcal{I}_\mathcal{R}} t_i^\bullet$ is a $\text{SIC}_{f,\boldsymbol{x}}$.*

Proof. Let $\boldsymbol{x}$ be a set of variables, and let $\mathcal{R}$ be a relation of absorption with support $\mathcal{I}_\mathcal{R}$ for f. For every $i \in \mathcal{I}_\mathcal{R}$ let $t_i^\bullet$ be a $\text{SIC}_{t_i,\boldsymbol{x}}$. By definition $t_i^\bullet(\boldsymbol{a}) \models \forall \boldsymbol{x}.\forall \boldsymbol{y}.t\,(\boldsymbol{x}, \boldsymbol{a}) = t\,(\boldsymbol{y}, \boldsymbol{a})$, and therefore $\bigwedge_{i\in\mathcal{I}_\mathcal{R}} t_i^\bullet(\boldsymbol{a}) \models \bigwedge_{i\in\mathcal{I}_\mathcal{R}} \forall \boldsymbol{x}.\forall \boldsymbol{y}.t_i\,(\boldsymbol{x}, \boldsymbol{a}) = t_i\,(\boldsymbol{y}, \boldsymbol{a})$. Finally, as $\mathcal{R}$ is a relation of absorption for f, $\mathcal{R}\,(t_i\,(\boldsymbol{a}))\bigwedge_{i\in\mathcal{I}_\mathcal{R}} t_i^\bullet(\boldsymbol{a}) \models \forall \boldsymbol{x}.\forall \boldsymbol{y}.f\,(t_i\,(\boldsymbol{x}, \boldsymbol{a})) = f\,(t_i\,(\boldsymbol{y}, \boldsymbol{a}))$, i.e., $\mathcal{R}\left(t_{i\in\mathcal{I}_\mathcal{R}}\right)\bigwedge_{i\in\mathcal{I}_\mathcal{R}} t_i^\bullet$ is a $\text{SIC}_{f,\boldsymbol{x}}$. □

Previous examples (Section 5.5.1) can be recast in term of $\mathcal{R}$-absorbing function, as shown in Figure 5.3, proving their correctness. Note that regarding our end-goal, we should accept only $\mathcal{R}$-absorbing functions in QF-$\mathcal{T}$.

5.6 Implementation and Experimental Evaluation

This section describes the implementation of our method (Section 5.6.1) for bitvectors and arrays (ABV), together with experimental evaluation (Section 5.6.2). Our prototype Tfml (*Taint engine for ForMuLa*) comprises 7 klocs of OCaml. Its code is open source and available online[2], together with all the benchmarks we use for experimental evaluation.

[2] http://benjamin.farinier.org/cav2018/

5.6.1 Implementation

Given an input formula in the SMT-LIB format [BST10] (ABV theory), Tfml performs several normalizations before adding taint information following Algorithm 5.1. The process ends with simplifications as taint usually introduces many constant values, and a new SMT-LIB formula is output. In order to make the whole procedure feasible in practice, Tfml makes use in addition of the following technicalities:

Maximal Sharing This stage is crucial as it allows to avoid term duplication in `theorySIC` (Algorithm 5.2, Section 5.4.3, Proposition 5.4). To this end, Tfml uses hash-consing [FC06] to achieve in-memory maximal sharing for syntactically equal terms. As a side effect, maximal sharing permits to check terms syntactic equality in constant time, as it makes syntactic equality equivalent to physical equality. When the final formulas is output, we introduce new names via *let-binding* for relevant sub-terms in order to share their textual representation and keep the formula as concise as possible.

Simplifications We perform constant propagation and rewriting (standard rules, e.g. $x - x \mapsto 0$ or $x \times 1 \mapsto x$) on both initial and transformed formulas – equality is soundly approximated by syntactic equality. These simplifications are studied in detail in Chapter 6, Section 6.4.2.

Shadow Arrays We encode taint constraints over arrays through *shadow arrays*. For each array declared in the formula, we declare a (taint) shadow array. The default value for all cells of the shadow array is the taint of the original array, and for each value stored (resp. read) in the original array, we store (resp. read) the taint of the value in the shadow array. As logical arrays are infinite, we cannot constrain all the values contained in the initial shadow array. Instead, we rely on a common trick in array theory: we constrain only cells corresponding to a relevant read index in the formula.

Iterative Skolemization While we have supposed along the chapter to work on skolemized formulas, we have to be more careful in practice. Indeed, skolemization introduce dependencies between a skolemized variable and all its preceding universally quantified variables, blurring our analysis and likely resulting in considering the whole formula as dependent. Instead, we follow an iterative process: 1) Skolemize the first block of existentially quantified variables; 2) Compute the independence condition for any targeted variable in the first block of universal quantifiers and remove these quantifiers; 3) Repeat. This results in

full Skolemization together with the construction of an independence condition, while avoiding many unnecessary dependencies.

5.6.2 Evaluation

We experimentally evaluate the following research questions: *RQ1* How does our approach perform with regard to state-of-the-art approaches for model generation of quantified formulas? *RQ2* How effective is it at lifting quantifier-free solvers into (SAT-only) quantified solvers? *RQ3* How efficient is it in terms of preprocessing time and formula size overhead? We evaluate our method on a set of formulas combining arrays and bitvectors (paramount in software verification), against state-of-the-art solvers for these theories.

Protocol The experimental setup below runs on an Intel(R) Xeon(R) E5-2660 v3 @ 2.60GHz, 4GB RAM per process, and a TIMEOUT of 1 000s per formula.

Metrics For *RQ1* we compare the number of SAT and UNKNOWN answers between solvers supporting quantification, with and without our approach. For *RQ2*, we compare the number of SAT and UNKNOWN answers between quantifier-free solvers enhanced by our approach and solvers supporting quantification. For *RQ3*, we measure preprocessing time and formulas size overhead.

Benchmarks We consider two sets of ABV formulas. First, a set of 1 421 formulas from (a modified version of) the Symbolic Execution tool BINSEC [DBT+16] representing quantified reachability queries (cf. Section 5.2) over BINSEC benchmark programs (security challenges, e.g. `crackme` or vulnerability finding). The initial (array) memory is quantified so that models depend only on user input. Second, a set of 1 269 ABV formulas generated from formulas of the QF-ABV category of SMT-LIB [BST10] – sub-categories `brummayerbiere`, `dwp formulas` and `klee selected`. The generation process consists in universally quantifying some of the initial array variables, mimicking quantified reachability problems.

Competitors For *RQ1*, we compete against the two state-of-the-art SMT solvers for quantified formulas CVC4 [BCD+11] (finite model instantiation [RTGK13]) and Z3 [dMB08] (model-based instantiation [GdM09]). We also consider degraded versions CVC4_E and Z3_E that roughly represent standard E-matching [DNS05]. For *RQ2* we use Boolector (Btor) [BB09], one of the very best QF-ABVsolvers.

Table 5.1 – Answers and resolution time (in seconds, include TIMEOUT).

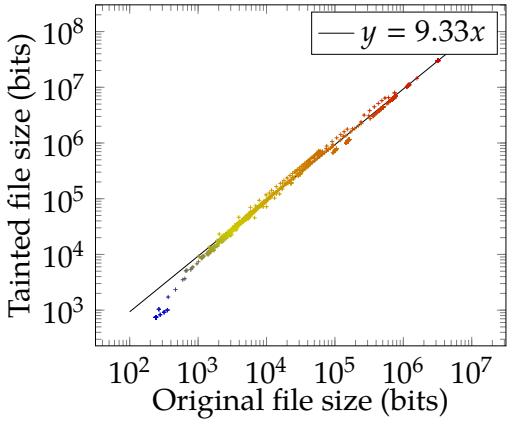
			Btor•	CVC4	CVC4•	CVC4$_E$	CVC4$_E$•	Z3	Z3•	Z3$_E$	Z3$_E$•
SMT-LIB	#	SAT	**399**	84	242	84	242	261	366	87	366
		UNSAT	N/A	0	N/A	0	N/A	165	N/A	0	N/A
		UNKNOWN	870	1 185	1 027	1 185	1 027	843	903	1 182	903
	total time		**349**	165	194 667	165	196 934	270 150	36 480	192	41 935
BINSEC	#	SAT	**1 042**	951	954	951	954	953	**1 042**	953	**1 042**
		UNSAT	N/A	62	N/A	62	N/A	319	N/A	62	N/A
		UNKNOWN	379	408	467	408	467	149	379	406	379
	total time		**1 152**	64 761	76 811	64 772	77 009	30 235	11 415	135	11 604

solver•: solver enhanced with our method Z3$_E$, CVC4$_E$: essentially E-matching

Table 5.2 – Complementarity of our approach with existing solvers (SAT instances).

		CVC4•			Z3•			Btor•		
SMT-LIB	CVC4	-10	+168	[252]				-10	+325	[409]
	Z3				-119	+224	[485]	-86	+224	[485]
BINSEC	CVC4	-25	+28	[979]				-25	+116	[1 067]
	Z3				-25	+114	[1 067]	-25	+114	[1 067]

Results Table 5.1, Table 5.2 and Figure 5.4 sum up our experimental results, which have all been cross-checked for consistency. Table 5.1 reports the number of successes (SAT or UNSAT) and failures (UNKNOWN), plus total solving times. The • sign indicates formulas preprocessed with our approach. In that case it is impossible to correctly answer UNSAT (no WIC checking), the UNSAT line is thus N/A. Since Boolector does not support quantified ABV formulas, we only give results with our approach enabled. Table 5.1 reads as follows: of the 1 269 SMT-LIB formulas, standalone Z3 solves 426 formulas (261 SAT, 165 UNSAT), and 366 (all SAT) if preprocessed. Interestingly, our approach always improves the underlying solver in terms of solved (SAT) in-



Maximal size ratio	12.48
Minimal size ratio	2.81
Average size ratio	8.73
Standard deviation	0.78

Figure 5.4 – Overhead in formula size.

Table 5.3 – GRUB example.

		Btor•	Z3
#	SAT	**540**	1
	UNSAT	N/A	42
	UNKNOWN	355	852
total time		**16 732**	159 765

Table 5.4 – Best approaches.

		former	new	
		Z3	Btor•	Btor• ▹ Z3
SMT-LIB	SAT	261	399	485
	UNSAT	165	N/A	165
	UNKNOWN	843	870	619
	time	270 150	350	94 610
BINSEC	SAT	953	1 042	1 067
	UNSAT	319	N/A	319
	UNKNOWN	149	379	35
	time	64 761	1 152	1 169

stances, either in a significant way (SMT-LIB) or in a modest way (BINSEC). Yet, recall that in a software verification setting every win matters (possibly new bug found or new assertion proved). For Z3•, it also strongly reduces computation time. Last but not least, Boolector• (a pure QF-solver) turns out to have the best performance on SAT-instances, beating state-of-the-art approaches both in terms of solved instances and computation time.

Table 5.2 substantiates the complementarity of the different methods, and reads as follow: for SMT-LIB, Boolector• solves 224 (SAT) formulas missed by Z3, while Z3 solves 86 (SAT) formulas missed by Boolector•, and 485 (SAT) formulas are solved by either one of them.

Figure 5.4 shows formula size averaging a 9-fold increase (min 3, max 12): yet they are easier to solve because they are more constrained. Regarding performance and overhead of the tainting process, *taint time is almost always less than 1s* in our experiments (not shown here), 4min for worst case, clearly dominated by resolution time. The worst case is due to a pass of linearithmic complexity which can be optimized to be logarithmic.

Pearls We show hereafter two particular applications of our method. Table 5.3 reports results of another Symbolic Execution experiment, on the `grub` example. On this example, Boolector• completely outperforms existing approaches. As a second application, while the main drawback of our method is that it precludes proving UNSAT, this is easily mitigated by complementing the approach with another one geared (or able) to proving UNSAT, yielding efficient solvers for quantified formulas, as shown in Table 5.4.

Conclusion Experiments demonstrate the relevance of our taint-based technique for model generation. (*RQ1*) Results in Table 5.1 shows that our approach greatly facilitates the resolution process. *On these examples, our method performs better than state-of-the-art solvers but also strongly complements them (Table 5.2)*. (*RQ2*) Moreover, Table 5.1 demonstrates that our technique is highly effective at lifting quantifier-free solvers to quantified formulas, in both number of SAT answers and computation time. *Indeed, once lifted, Boolector performs better (for SAT-only) than Z3 or CVC4 with full quantifier support*. Finally (*RQ3*) our tainting method itself is very efficient both in time and space, making it perfect either for a preprocessing step or for a deeper integration into a solver. In our current prototype implementation, we consider the cost to be low.

5.7 Related Works

Traditional approaches for solving quantified formulas essentially involve either generic methods geared to prove unsatisfiability and validity [DNS05], or complete but dedicated approaches for particular theories [BMS06, WHdM10]. Besides, some recent methods [IJS08, GdM09, RTGK13] aim to be correct and complete for larger classes of theories.

Generic Method for Unsatisfiability Broadly speaking, these methods iteratively instantiate axioms until a contradiction is found. They are generic with regard to the underlying theory and allow to reuse standard theory solvers, but termination is not guaranteed. Also, they are more suited to prove unsatisfiability than to find models. In this family, E-matching [DNS05, dMB07] shows reasonable cost when combined with conflict-based instantiation [RTdM14] or semantic triggers [DCKP12, DCKP16]. In pure first-order logic (without theories), quantifiers are mainly handled through resolution and superposition [BG94, NR01] as done in Vampire [RV02, KV13] and E [Sch02].

Complete Methods for Specific Theories Much work has been done on designing complete decision procedures for quantified theories of interest, notably array properties [BMS06], quantified theory of bitvectors [WHdM10, JS16], Presburger arithmetic or Real Linear Arithmetic [BKRW11, FK16]. Yet, they usually come at a high cost.

Generic Methods for Model Generation Some recent works detail attempts at more general approaches to model generation.

Local theory extensions [IJS08, BRK+15] provide means to extend some decidable theories with free symbols and quantifications, retaining decidability. The approach identifies specific forms of formulas and quantifications (bounded), such that these theory extensions can be solved using finite instantiation of quantifiers together with a decision procedure for the original theory. The main drawback is that the formula size can increase a lot.

Model-based quantifier instantiation is an active area of research notably developed in Z3 and CVC4. The basic line is to consider the partial model under construction in order to find the right quantifier instantiations, typically in a try-and-refine manner. Depending on the variants, these methods favor either satisfiability or unsatisfiability. They build on the underlying quantifier-free solver and can be mixed with E-matching techniques, yet each refinement yields a solver call and the refinement process may not terminate. Ge and de Moura [GdM09] study decidable fragments of first-order logic modulo theories for which model-based quantifier instantiation yields soundness and refutational completeness. Reynolds *et al.* [RTdM14], Barbosa [Bar16] and Preiner *et al.* [PNB17] use models to guide the instantiation process towards instances refuting the current model. *Finite model quantifier instantiation* [RTGK13, RTG+13] reduces the search to finite models, and is indeed geared toward model generation rather than unsatisfiability. Similar techniques have been used in program synthesis [RDK+15].

We drop support for the unsatisfiable case but get more flexibility: we deal with quantifiers on any sort, the approach terminates and is lightweight, in the sense that it requires a single call to the underlying quantifier-free solver.

Other Our method can be seen as taking inspiration from program taint analysis [DD77, Ørb95] developed for checking the non-interference [Smi07] of public and secrete input in security-sensitive programs. As far as the analogy goes, our approach should not be seen as checking non-interference, but rather as inferring preconditions of non-interference. Moreover, our formula-tainting technique is closer to dynamic program-tainting than to static program-tainting, in the sense that precise dependency conditions are statically inserted at preprocess-time, then precisely explored at solving-time.

Finally, Darvas *et al.* [DMR08] presents a bottom-up formula strengthening method. Their goal differ from ours, as they are interested in formula well-definedness (rather than independence) and validity (rather than model generation).

5.8 Conclusion

This chapter addressed the problem of generating models of quantified first-order formulas over built-in theories. We proposed a correct and generic approach based on a reduction to the quantifier-free case through the inference of independence conditions. The technique is applicable to any theory with a decidable quantifier-free case and allows to reuse all the work done on quantifier-free solvers. The method significantly enhances the performances of state-of-the-art SMT solvers for the quantified case, and supplements the latest advances in the field. Chapter 7 will put into practice our taint-based elimination on quantified formulas generated by Symbolic Execution.

Future developments aim to tackle the definition of more precise inference mechanisms of independence conditions, the identification of interesting subclasses for which inferring weakest independence conditions is feasible, and the combination with other quantifier instantiation techniques.

Bibliography

[Bar16] Haniel Barbosa. Efficient Instantiation Techniques in SMT (work in progress). In *Proceedings of the 5th Workshop on Practical Aspects of Automated Reasoning co-located with International Joint Conference on Automated Reasoning (IJCAR 2016), Coimbra, Portugal, July 2nd, 2016.*, pages 1–10, 2016.

[BB09] Robert Brummayer and Armin Biere. Boolector: An Efficient SMT Solver for Bit-Vectors and Arrays. In *Tools and Algorithms for the Construction and Analysis of Systems, 15th International Conference, TACAS 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings*, pages 174–177, 2009.

[BCD+11] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, pages 171–177, 2011.

[BG94] Leo Bachmair and Harald Ganzinger. Rewrite-Based Equational Theorem Proving with Selection and Simplification. *J. Log. Comput.*, 4(3):217–247, 1994.

[Bie09] Armin Biere. Bounded Model Checking. In *Handbook of Satisfiability*, pages 457–481. 2009.

[BKRW11] Angelo Brillout, Daniel Kroening, Philipp Rümmer, and Thomas Wahl. Beyond Quantifier-Free Interpolation in Extensions of Presburger Arithmetic. In *Verification, Model Checking, and Abstract Interpretation - 12th International Conference, VMCAI 2011, Austin, TX, USA, January 23-25, 2011. Proceedings*, pages 88–102, 2011.

[BMS06] Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. What's decidable about arrays? In *Verification, Model Checking, and Abstract Interpretation, 7th International Conference, VMCAI 2006, Charleston, SC, USA, January 8-10, 2006, Proceedings*, pages 427–442, 2006.

[BRK+15] Kshitij Bansal, Andrew Reynolds, Tim King, Clark W. Barrett, and Thomas Wies. Deciding local theory extensions via e-matching. In *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part II*, pages 87–105, 2015.

[BSST09] Clark W. Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. Satisfiability Modulo Theories. In *Handbook of Satisfiability*, pages 825–885. 2009.

[BST10] Clark Barrett, Aaron Stump, and Cesare Tinelli. The SMT-LIB Standard: Version 2.0. In A. Gupta and D. Kroening, editors, *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, UK)*, 2010.

[DBT+16] Robin David, Sébastien Bardin, Thanh Dinh Ta, Laurent Mounier, Josselin Feist, Marie-Laure Potet, and Jean-Yves Marion. BINSEC/SE: A Dynamic Symbolic Execution Toolkit for Binary-Level Analysis. In *IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering, SANER 2016, Osaka, Japan, March 14-18, 2016 - Volume 1*, pages 653–656, 2016.

[DCKP12] Claire Dross, Sylvain Conchon, Johannes Kanig, and Andrei Paskevich. Reasoning with Triggers. In *10th International Workshop on Satisfiability Modulo Theories, SMT 2012, Manchester, UK, June 30 - July 1, 2012*, pages 22–31, 2012.

[DCKP16] Claire Dross, Sylvain Conchon, Johannes Kanig, and Andrei Paskevich. Adding Decision Procedures to SMT Solvers Using Axioms with Triggers. *J. Autom. Reasoning*, 56(4):387–457, 2016.

[DD77] Dorothy E. Denning and Peter J. Denning. Certification of Programs for Secure Information Flow. *Commun. ACM*, 20(7):504–513, 1977.

[dMB07] Leonardo Mendonça de Moura and Nikolaj Bjørner. Efficient e-matching for SMT solvers. In *Automated Deduction - CADE-21, 21st International Conference on Automated Deduction, Bremen, Germany, July 17-20, 2007, Proceedings*, pages 183–198, 2007.

[dMB08] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, pages 337–340, 2008.

[DMR08] Ádám Darvas, Farhad Mehta, and Arsenii Rudich. Efficient Well-Definedness Checking. In *Automated Reasoning, 4th International Joint Conference, IJCAR 2008, Sydney, Australia, August 12-15, 2008, Proceedings*, pages 100–115, 2008.

[DNS05] David Detlefs, Greg Nelson, and James B. Saxe. Simplify: a theorem prover for program checking. *J. ACM*, 52(3):365–473, 2005.

[FBBP18] Benjamin Farinier, Sébastien Bardin, Richard Bonichon, and Marie-Laure Potet. Model generation for quantified formulas: A taint-based approach. In *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part II*, pages 294–313, 2018.

[FC06] Jean-Christophe Filliâtre and Sylvain Conchon. Type-safe modular hash-consing. In *Proceedings of the ACM Workshop on ML, 2006, Portland, Oregon, USA, September 16, 2006*, pages 12–19, 2006.

[FK16] Azadeh Farzan and Zachary Kincaid. Linear Arithmetic Satisfiability via Strategy Improvement. In *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence, IJCAI 2016, New York, NY, USA, 9-15 July 2016*, pages 735–743, 2016.

[GdM09] Yeting Ge and Leonardo Mendonça de Moura. Complete instantiation for quantified formulas in satisfiabiliby modulo theories. In *Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings*, pages 306–320, 2009.

[GLM12] Patrice Godefroid, Michael Y. Levin, and David A. Molnar. SAGE: Whitebox Fuzzing for Security Testing. *ACM Queue*, 10(1):20, 2012.

[IJS08] Carsten Ihlemann, Swen Jacobs, and Viorica Sofronie-Stokkermans. On local reasoning in verification. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, pages 265–281, 2008.

[JS16] Martin Jonás and Jan Strejcek. Solving Quantified Bit-Vector Formulas Using Binary Decision Diagrams. In *Theory and Applications of Satisfiability Testing - SAT 2016 - 19th International Conference, Bordeaux, France, July 5-8, 2016, Proceedings*, pages 267–283, 2016.

[KS08] Daniel Kroening and Ofer Strichman. *Decision Procedures - An Algorithmic Point of View*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2008.

[KV13] Laura Kovács and Andrei Voronkov. First-Order Theorem Proving and Vampire. In *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, pages 1–35, 2013.

[NR01] Robert Nieuwenhuis and Albert Rubio. Paramodulation-Based Theorem Proving. In *Handbook of Automated Reasoning (in 2 volumes)*, pages 371–443. 2001.

[Ørb95] Peter Ørbæk. Can you Trust your Data? In *TAPSOFT'95: Theory and Practice of Software Development, 6th International Joint Conference CAAP/FASE, Aarhus, Denmark, May 22-26, 1995, Proceedings*, pages 575–589, 1995.

[PNB17] Mathias Preiner, Aina Niemetz, and Armin Biere. Counterexample-Guided Model Synthesis. In *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part I*, pages 264–280, 2017.

[RDK+15] Andrew Reynolds, Morgan Deters, Viktor Kuncak, Cesare Tinelli, and Clark W. Barrett. Counterexample-guided quantifier instantiation for synthesis in SMT. In *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part II*, pages 198–216, 2015.

[RL18] Antoine Rollet and Arnaud Lanoix, editors. *Approches Formelles dans l'Assistance au Développement de Logiciels, AFADL 2018, Grenoble, France, June 13-15, 2018*, 2018.

[RTdM14] Andrew Reynolds, Cesare Tinelli, and Leonardo Mendonça de Moura. Finding conflicting instances of quantified formulas in SMT. In *Formal Methods in Computer-Aided Design, FMCAD 2014, Lausanne, Switzerland, October 21-24, 2014*, pages 195–202, 2014.

[RTG+13] Andrew Reynolds, Cesare Tinelli, Amit Goel, Sava Krstic, Morgan Deters, and Clark Barrett. Quantifier instantiation techniques for finite model finding in SMT. In *Automated Deduction - CADE-24 - 24th International Conference on Automated Deduction, Lake Placid, NY, USA, June 9-14, 2013. Proceedings*, pages 377–391, 2013.

[RTGK13] Andrew Reynolds, Cesare Tinelli, Amit Goel, and Sava Krstic. Finite model finding in SMT. In *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, pages 640–655, 2013.

[RV02] Alexandre Riazanov and Andrei Voronkov. The design and implementation of VAMPIRE. *AI Commun.*, 15(2-3):91–110, 2002.

[Sch02] Stephan Schulz. E - a brainiac theorem prover. *AI Commun.*, 15(2-3):111–126, 2002.

[Smi07] Geoffrey Smith. Principles of Secure Information Flow Analysis. In *Malware Detection*, pages 291–307. 2007.

[WHdM10] Christoph M. Wintersteiger, Youssef Hamadi, and Leonardo Mendonça de Moura. Efficiently solving quantified bit-vector formulas. In *Proceedings of 10th International Conference on Formal Methods in Computer-Aided Design, FMCAD 2010, Lugano, Switzerland, October 20-23*, pages 239–246, 2010.

Chapter 6

Arrays Made Simpler: An Efficient, Scalable and Thorough Preprocessing

The theory of arrays has a central place in software verification due to its ability to model data structures or memory, as done in Chapter 4. Yet, this theory is known to be hard to solve in both theory and practice, especially in the case of very long formulas coming from unrolling-based verification methods, like those obtained in Chapter 2 with binary-level Symbolic Execution. Standard simplification techniques *à la read-over-write* suffer from two main drawbacks: they do not scale on very long sequences of stores and they miss many simplification opportunities because of a crude syntactic (dis-)equality reasoning. In this chapter we propose a new approach to array formula simplification based on a new dedicated data structure together with original simplifications and low-cost reasoning. The technique is efficient, scalable and it yields significant simplification. The impact on formula resolution is always positive, and it can be dramatic on some specific classes of problems of interest, e.g. very long formula or binary-level Symbolic Execution. While currently implemented as a preprocessing, the approach would benefit from a deeper integration in an array solver. *The content of this chapter was presented in [BM18] and published in [FDBL18].*

6.1 Introduction

Context Automatic decision procedures for Satisfiability Modulo Theory [BT18] are at the heart of almost all recent formal verification methods [CKL04, BM07, CS13, Rus05]. Especially, the *theory of arrays* we presented in Chapter 3, Section 3.3.3, enjoys a central position in software verification as it allows to model memory or essential data structures

$$\begin{array}{ll} \cdot[\cdot] : \text{Array}\ \mathcal{I}\ \mathcal{E} \to \mathcal{I} \to \mathcal{E} & \forall a\ i\ e.\ (a[i] \leftarrow e)[i] = e \\ \cdot[\cdot] \leftarrow \cdot : \text{Array}\ \mathcal{I}\ \mathcal{E} \to \mathcal{I} \to \mathcal{E} \to \text{Array}\ \mathcal{I}\ \mathcal{E} & \forall a\ i\ j\ e.\ (i \neq j) \Rightarrow (a[i] \leftarrow e)[j] = a[j] \end{array}$$

Figure 6.1 – The theory of arrays (ROW-axioms).

such as maps, vectors and hash tables.

We recall that, given a set $\mathcal{I}$ of indexes and a set $\mathcal{E}$ of elements, the theory of arrays describes the set Array $\mathcal{I}$ $\mathcal{E}$ of all arrays mapping each index $i \in \mathcal{I}$ to an element $e \in \mathcal{E}$. Actually, logical arrays can be seen as infinite updatable maps implicitly defined by a succession of writes from an initial map. These arrays are defined by the two operations read $(\cdot[\cdot]\,\cdot)$ and write $(\cdot[\cdot] \leftarrow \cdot)$, whose semantic is given in Figure 6.1 by so-called read-over-write axioms (ROW-axioms).

Despite its simplicity, the satisfiability problem for the theory of arrays is NP-complete[1]. Indeed, it implies deciding (dis-)equalities between read and written indexes on *read-over-write* terms (ROW) of the form $(a[i] \leftarrow e)[j]$, potentially yielding nested case-splits. Standard decision procedures for arrays consist in eliminating as much ROW as possible through a preprocessing step [GD07], using axioms from Figure 6.1 as rewriting rules, and then enumerating all possible (dis-)equalities in ROW, yielding a potentially huge search space — the remaining ROW-axioms can be introduced lazily to mitigate this issue [BB09b].

Problem and Challenge Yet, this is not satisfactory when considering very long chains of writes, as can be encountered in unfolding-based verification techniques such as Symbolic Execution (SE) [CS13] or Bounded Model checking [CKL04] — the case of Deductive Verification is different since user-defined invariants prevent the unfolding. The theory of arrays can then quickly become a bottleneck of constraint solving. Especially, the ROW-simplification step is often very limited, for two reasons. **First**, exploring for every read in a backward manner the corresponding list of all writes yields a *quadratic time cost* (in the number of array operations) and therefore *it does not scale to very long formulas*. This is a major issue in practice as, for example, Symbolic Execution over malware or obfuscated programs [SBP18, BDM17, YJWD15] may have to consider execution traces of several millions of instructions, yielding formulas with several hundreds of thousands of array operations. Note also that bounding the backward exploration misses too many ROW-simplifications. **Second**, *(dis)-equalities*

[1]Reduction of the program equivalence problem in presence of arrays (sequential, boolean case) to the equivalence case without arrays but with **if** · **then** · **else** · operators, to SAT [DS78].

can be rarely decided during preprocessing as standard methods rely on efficient but *crude approximate equality checks* (typically, syntactic term equality), limiting again the power of these approaches. With such checks, index equality may be sometimes proven, but disequality can never be — except in the very restricted case of constant-value indexes.

Proposal We present a novel approach to ROW-simplification named FAS (*Fast Array Simplification*), allowing to scale and to simplify much more ROW than previous approaches. The technique is based on three key components:

- A re-encoding of write sequences (total order) as sequences of packs of independent writes (partial order), together with a dedicated data structure (*map list*) ensuring scalability;
- A new simple normalization step (*base normalization*) allowing to amplify the efficiency of syntactic (dis-)equality checks;
- A lightweight integration of *domain-based reasoning* over packs yielding even more successful (dis-)equality checks for only a slight overhead.

Experimental results demonstrate that FAS scales over very large formulas (several hundreds of thousands of ROW) typically coming from Symbolic Execution and can yield very significant gains in terms of runtime — possibly passing from hours to seconds.

Contributions This chapter makes the following contributions:

- We present in detail the new FAS preprocessing step for scalable and thorough array constraint simplification (Section 6.4), along with its three key components: dedicated data structure (Section 6.4.1), base normalization (Section 6.4.2) and domain reasoning (Section 6.4.4);
- We experimentally evaluate FAS in different settings for three leading SMT solvers (Section 6.5). The technique is fast and scalable, it yields a significant reduction of the number of ROW with always a positive impact on resolution time. This impact is even dramatic for some key usage scenarios such as SE-like formulas with small TIMEOUT or very large size.

Discussion In our view, FAS reaches a sweet spot between efficiency and impact on resolution. Experiments demonstrate that even major solvers benefit from it, with gains ranging from slight to very high depending on the setting. While presented as a

$\texttt{esp}_0$: BitVec 16
$\texttt{mem}_0$: Array BitVec 16 BitVec 16

assert $(\texttt{esp}_0 > 61440)$
$\texttt{mem}_1 \triangleq \texttt{mem}_0[\texttt{esp}_0 - 16] \leftarrow 1415$
$\texttt{esp}_1 \triangleq \texttt{esp}_0 - 64$
$\texttt{eax}_0 \triangleq \texttt{mem}_1[\texttt{esp}_1 + 48]$
assert $(\texttt{mem}_1[\texttt{eax}_0] = 9265)$

$\texttt{esp}_0$: BitVec 16
$\texttt{mem}_0$: Array BitVec 16 BitVec 16

assert $(\texttt{esp}_0 > 61440)$
assert $(\texttt{mem}_0[1415] = 9265)$

Figure 6.2 – A formula in the theory of arrays (left) and its simplification with FAS (right).

preprocessing, FAS would clearly benefit from a deeper integration inside an array solver, in order to take advantage of more simplification opportunities along the resolution process.

6.2 Motivation

Let us detail how the formula in the left part of Figure 6.2 can be simplified into the formula in the right part using our new FAS simplification procedure for arrays. We focus on the last assertion which involves a read on $\texttt{mem}_1 \triangleq \texttt{mem}_0[\texttt{esp}_0 - 16] \leftarrow 1415$, i.e. a *read-over-write*. Let us denote $i \triangleq \texttt{esp}_0 - 16$. The read occurs at index $\texttt{eax}_0$, which is itself the result of a read on $\texttt{mem}_1$ at index $j \triangleq \texttt{esp}_1 + 48$. According to arrays semantics (Figure 6.1), we must try to decide whether i and j (resp. $\texttt{eax}_0$ and i) are equal or different. *The standard syntactic equality check is not conclusive here*. But $\texttt{esp}_1 \triangleq \texttt{esp}_0 - 64$, therefore j can be rewritten into $\texttt{esp}_0 - 16$ (*base normalization* in FAS), which is exactly i. Hence $i = j$ is proven. By applying array axioms, we deduce that $\texttt{eax}_0 \triangleq 1415$, and the last assertion becomes $\texttt{mem}_1[1415] = 9265$. We now try to decide whether i and 1415 are equal or different. *Again, the standard syntactic equality check fails*. Yet, by the first assertion we deduce that $i > 61424$ (*domain propagation* in FAS), leading to $i \neq 1415$. Therefore $\texttt{mem}_1$ is safely replaced by $\texttt{mem}_0$ in the last assertion which becomes $\texttt{mem}_0[1415] = 9265$. Finally, as assertions in the formula now only refer to $\texttt{esp}_0$ and $\texttt{mem}_0$, we erase all the intermediate definitions to obtain the simplified formula.

This little mental gymnastic emphasises two important aspects of ROW-simplifications. First, simplifications often require (dis-)equality reasoning beyond pure syntactic equality. Second, simplifications involve a backward reasoning through the formula which may become prohibitive on large formulas if not treated with care (not shown

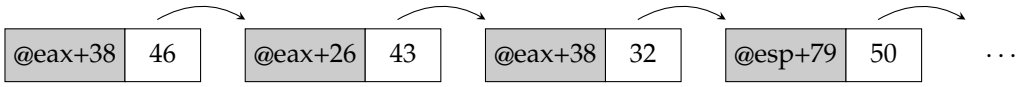


Figure 6.3 – Arrays represented as sequences (lists) of writes.

here, up to 1h simplification time in Figure 6.10). *Our proposal focuses especially on these two aspects.*

6.3 Standard Simplifications for *read-over-write*

The theory of arrays has been introduced in Chapter 3, Section 3.3.3, and in Figure 6.1. As already stated, the main difficulty for reasoning over arrays comes from terms of the form $(a[i] \leftarrow e)[j]$, called *read-over-write* (ROW), since depending on whether $i = j$ holds or not, the term evaluates to e (SELECT-HIT) or to $a[j]$ (SELECT-MISS). *Array (formula) simplification* consists in removing as many ROW as possible before resolution by proving (when possible) the validity of the (dis-) equality of such pairs of indexes (i, j) and rewrite the term accordingly. Such simplification procedures critically depend on two factors: 1) the **equality check procedure**, and 2) the **underlying representation of an array** and its revisions arising from successive writes.

The **equality check** must be both *efficient* — simplifying a formula must be cheaper than solving it, and *correct* — all proven (dis-)equalities must indeed hold. It can thus only be *approximated*, i.e. it is incomplete and may miss some valid (dis-)equalities. The standard solution is to rely on *syntactic term equality checking*. Obviously this is a *crude approximation*: disequality can never be proven (but for constant-value indexes), and as exemplified in Section 6.2, small syntactic variations of the same value can hinder proving equalities.

We present now two (unsatisfactory) standard **array representations**, coming either from the decision procedure community (the *list* representation: *generic but slow*) or from the Symbolic Execution community (the *map* representation: *efficient but restricted*).

Arrays Represented as Lists The standard representation of an array and its subsequent revisions is basically a "store-chain", the linked list of all successive writes in the array. Hence a fresh array is simply an empty list, while the array obtained by writing an element e at index i in array A is represented by a node containing (i, e) and pointing to the list representing A. Figure 6.3 illustrates this encoding. This approach is very generic — it can cope with symbolic indexes, and it is the one implicitly used

inside array solvers. In order to simplify a read at index j on array A, one must decide whether $i = j$ is valid for the pair (i, e) inside the head of the list representing A. If we succeed, then we can apply the ROW axiom and replace the read by value e. Otherwise, we try to decide whether $i \neq j$ is valid. If this is the case, then we use the second ROW axiom and move backward along the linked list. If not, the simplification process stops.

An inherent problem with this representation is the increase in the simplification cost as the number of writes rises. As mentioned in Section 6.2, this cost becomes prohibitive when dealing with large formulas. Indeed, one might be forced for each read to fully explore the write-list backward, yielding a *quadratic worst case time cost*. This is especially unfortunate because this worst case arises in situations where the simplification could perform the best, e.g. when all disequalities between indexes hold so that all reads could be replaced with accesses to the initial array (no more ROW). A workaround is to bound the backward exploration of the write-list, which reduces the worst case time cost to linear, but at the expense of limited simplifications (Figure 6.10, Section 6.5.4).

Arrays Represented as Maps In the restricted case where all indexes of reads and writes are constant values, a persistent map with logarithmic lookup and insertion can be used to simplify all ROW occurrences — yielding fast and scalable simplification. This representation is used in Symbolic Execution tools [CS13] with strong concretization policy [GKS05, DBF+16] during the formula generation step in order to limit the introduction of arrays, but it is not suited to general purpose array solvers as *it cannot cope with symbolic indexes*.

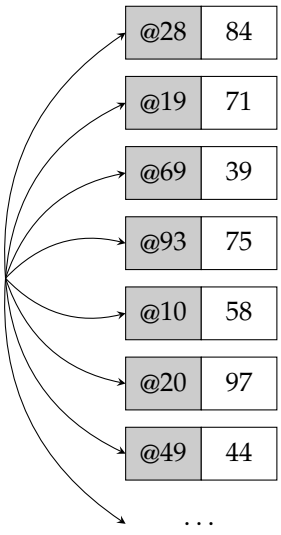


Figure 6.4 – Arrays represented as maps of writes (constant write indexes).

Here, a freshly declared array is represented by an empty map where indexes and elements sorts correspond to those of the array, and the array obtained after a write of element e at index i is simply represented by the map of the written array in which e is added at index i, as illustrated in Figure 6.4. Then the simplification of a read at index j becomes its substitution by the element mapped to j. In the case where no such element is found, the read occurs on the initial array. Therefore, we can either replace the array by the initial one or replace the read by a fresh symbol. In the latter case, we have to ensure that two reads are replaced by the same symbol if and only if they occur at the same index.

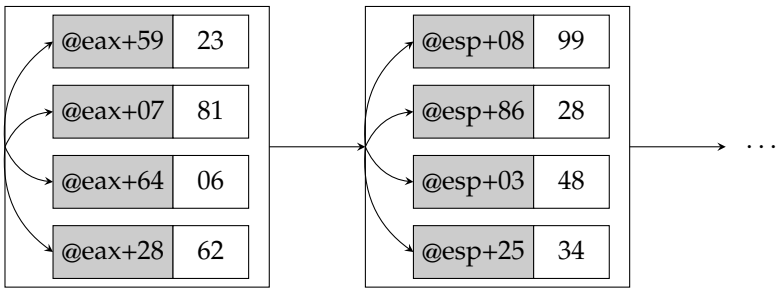


Figure 6.5 – Arrays represented as sequences of packs of independent writes (map list).

6.4 Efficient Simplification for *read-over-write*

We now present FAS (*Fast Array Simplification*), an efficient approach to *read-over-write* simplification. FAS combines three key ingredients: a new representation for arrays as a list of maps to ensure scalability, a dedicated rewriting step (base normalization) geared at improving the conclusiveness of syntactic (dis-)equality checks between indexes, and lightweight domain reasoning to go beyond purely syntactic checks.

6.4.1 Dedicated Data Structure: Arrays Represented as Lists of Maps

We look here for an array representation combining the advantages of the list representation (genericity) and the map representation (efficiency) presented in Section 6.3. As a preliminary remark, we can note that the map representation can be extended from the constant-indexes case to the case where all indexes of reads and writes are pairwise *comparable*. By comparable we mean that a binary comparison operator $\prec$ is defined and decidable for every pair of indexes in the formula. Yet, if such a hypothesis might sometimes be satisfied, it is not necessary the case, for example when indexes contain uninterpreted symbols.

The representation of arrays we propose, **lists of maps** (denoted *map lists*), aims precisely at combining advantages of maps when all indexes are pairwise comparable while being as general as lists in other situations. Our array representation can be thought of as a list of *packs of independent writes*. The idea is that sets of comparable (and proven different) indexes can be packed together into map-like data structures, allowing efficient (i.e. logarithmic) search on these packs of indexes during the application of ROW-like simplification rules. *While the idea is presented here in general, we instantiate it in Section 6.4.3, Figure 6.7, and in Section 6.4.4, Figure 6.8.*

In this representation, nodes of the list are maps from pairwise-comparable indexes to written elements, as illustrated in Figure 6.5. A **fresh array** is represented as an

empty list (of maps). The array obtained after the **write of element** e **at index** i is defined by:

- If i is comparable with all other indexes of elements already inserted in the map at head position, then we add the element e at index i into this map (STORE-HIT);
- Else we add to the list a fresh node containing the singleton map of index i to element e (STORE-MISS).

For a **read at index** j, the simplification of ROW is done as follows:

- If indexes in the head position map of the list representing the array are all comparable with j, then if j belongs to this map we substitute the read by the associated element (SELECT-HIT), else we re-iterate on the following node in the list (SELECT-MISS);
- Else, we abort (SELECT-ABORT).

A first version of the dedicated (dis-)equality checks we use is presented in Section 6.4.2. The whole FAS procedure, together with the associated notion of comparable, is formally described in Section 6.4.3, and a refinement using more semantic checks is presented in Section 6.4.4.

Intuitively, the benefit of this representation is that its behavior varies between the one of the list representation and the one of the map representation, depending on the proportion of indexes pairwise comparable. Indeed, when all indexes are pairwise comparable, the list only contains a single map of all indexes, which is equivalent to the map representation. And when none of the index pairs are comparable, the list is composed of singleton maps, which is equivalent to the list representation.

From a technical point of view, map lists enjoys several good properties:

Property 6.1 (Compactness)**.** By construction, all indexes in any map of a map list are pairwise comparable, while indexes from adjacent maps are never comparable.

Property 6.2 (Complexity)**.** Assuming that 1) we can decide efficiently (constant or logarithmic time) whether an index is comparable to all the other indexes of a given map, 2) that $<$ between comparable terms can also be efficiently decided (constant or logarithmic time), and 3) a decent implementation of maps (logarithmic time insertion and lookup), then:

- Array writes are computed in logarithmic time (map insertion) — where the standard list approach requires only constant time;

- Array reads are *also computed in logarithmic time* (map lookup) as SELECT-MISS can only led to SELECT-ABORT (Property 6.1) — where the standard list approach requires *linear time*.

In the case where all indexes are pairwise comparable, our representation contains a single map and simplification cost for r reads and w writes is bounded by $r \cdot ln(w)$, while the list approach requires a quadratic $r \cdot w$ time.

Finally, map lists allow to easily take into account some cases of *write-over-write* (a write masked by a later write at the same index can be ignored if no read happens in-between), while it requires a dedicated and expensive (w^2) treatment with lists.

6.4.2 Approximated Equality Check and Dedicated Rewriting

We consider as equality check a variation of syntactic term equality, namely **syntactic base/offset equality**, which is regarding two terms t_1 and t_2 defined as follows:

- If $t_1 \triangleq \beta_1 + \iota_1$ and $t_2 \triangleq \beta_2 + \iota_2$ — where β_1, β_2 are arbitrary terms (*bases*) and ι_1, ι_2 are constant values (*offsets*), and $\beta_1 = \beta_2$ (syntactically) then return the result of $\iota_1 = \iota_2$,

- Otherwise the check is not conclusive.

This equality check is *correct* and *efficient*, and it strictly extends syntactic term equality — the result is more often conclusive. Actually, in practice it turns out that this extension is significant. Indeed, a common pattern in array formulas coming from software analysis is reads or writes at indexes defined as the sum of a base and an offset (think of C or assembly programming idioms). Hence, dealing with such terms is particularly interesting for verification-oriented formulas.

Dedicated Rewriting: Base Normalization Yet, this equality check still suffers from the rigidity of syntactic approaches. Therefore it is worthwhile to normalizes indexes as much as possible by applying a dedicated set of rewriting rules called **base normalization** (*rebase*), cf. Figure 6.6. These rules are essentially based on limited inlining of variables together with associativity and commutativity rules of $+/-$ operators, the goal being to minimize the number of possible bases in order to increase the "conclusiveness" of our equality check, as done in example Section 6.2.

if $u \triangleq v$ then	$u + k$	$\rightsquigarrow$	$v + k$	alias inlining
if $u \triangleq v + l$ then	$u + k$	$\rightsquigarrow$	$v + (k + l)$	base/offset inlining
	$-(x + k)$	$\rightsquigarrow$	$(-k) - x$	constant negation
	$(x + k) + l$	$\rightsquigarrow$	$x + (k + l)$	constant packing
	$(x + k) + y$	$\rightsquigarrow$	$(x + y) + k$	constant lifting
	$(x + k) + (y + l)$	$\rightsquigarrow$	$(x + y) + (k + l)$	base/offset addition
	$(x + k) - (y + l)$	$\rightsquigarrow$	$(x - y) + (k - l)$	base/offset subtraction
		...		

Figure 6.6 – Example of base normalization rules. u, v are variables, k, l are constant values and x, y are terms. Non-inlining rules reduce either the number of operators or the depth of constant values, ensuring termination. Note that $(-k), (k + l), (k - l)$ are constant values, not terms.

Optimization: Sub-Term Sharing Sharing of sub-terms consists in giving a common name to two syntactically equal terms. This improvement is not new and was already used in Chapter 5, Section 5.6.1, but has an original implication is this context. Besides easing the decision of equality between terms, it remedies to an issue induced by the simplification of ROW. Indeed, the simplification of ROW can be seen as a kind of "inlining" stage, which may in some cases lead to terms size explosion. This problem arises when after a write of element e at index i, several reads at index i are simplified. It may result in numerous copies of term e, term which may contain itself other reads to simplify. By naming and sharing terms read and written in arrays, the sub-term sharing phase prevents this issue. *Experiments in Section 6.5.4 demonstrate the practical interest on very large formulas.*

6.4.3 The FAS Procedure

Using the generic algorithm of Section 6.4.1 with equality check and normalization from Section 6.4.2, we formalize FAS as the set of inference rules presented in Figure 6.7. Two terms will be said *comparable* when they share the same base β. STORE-HIT and STORE-MISS rules explain how to update the representation of an array on writes, and SELECT-HIT and SELECT-MISS rules explain how to simplify reads. STORE rules are presented as triples $\{\Lambda\}\, a[i] \leftarrow e\, \{\Lambda'\}$ where Λ' is the representation for $a[i] \leftarrow e$ when Λ is the representation for a. SELECT rules are presented as triples $\{\Lambda\} \vdash a[i] \rightsquigarrow e$ meaning that $a[i]$ can be rewritten in e when Λ is the representation for a.

$$\frac{i = \beta + \iota \qquad \iota \text{ a constant}}{\{\langle \Gamma, \beta, b\rangle :: \Lambda\}\, a[i] \leftarrow e \,\{\langle \Gamma[\iota] \leftarrow e, \beta, b\rangle :: \Lambda\}}\ \textsc{store-hit}$$

$$\frac{i = \alpha + \iota \qquad \alpha \neq \beta}{\{\langle \Gamma, \beta, b\rangle :: \Lambda\}\, a[i] \leftarrow e \,\{\langle \varnothing[\iota] \leftarrow e, \alpha, a\rangle :: \langle \Gamma, \beta, b\rangle :: \Lambda\}}\ \textsc{store-miss}$$

$$\frac{\Gamma[\iota] = e \qquad i = \beta + \iota}{\{\langle \Gamma, \beta, b\rangle :: \Lambda\} \vdash a[i] \rightsquigarrow e}\ \textsc{select-hit}$$

$$\frac{\{\Lambda\} \vdash b[i] \rightsquigarrow e \qquad \Gamma[\iota] = \varnothing \qquad i = \beta + \iota}{\{\langle \Gamma, \beta, b\rangle :: \Lambda\} \vdash a[i] \rightsquigarrow e}\ \textsc{select-miss}$$

Figure 6.7 – Inference rules for $\cdot[\cdot]$ and $\cdot[\cdot] \leftarrow \cdot$ using the *map list* representation.

The representation $\langle \Gamma, \beta, b\rangle :: \Lambda$ we use is a specialized version of the map list representation that we just defined, where Γ is a map, β is the common base of indexes present in Γ, b the last revision of the array written at a different index than β, and where Λ is the tail of the list. Assuming that all indexes have been normalized, if the base of the write index is equal to β, then the STORE-HIT rule applies and we add the written element into Γ. If the base of the write index is not equal to β, then the STORE-MISS rule applies. We add as a new node of the list a singleton map containing only the written element, the new base and the written array. For ROW-simplification, the SELECT-HIT rule states that if the base of the read index is equal to β, and if there is an element in Γ mapped to this index, then we return this element. Finally the SELECT-MISS rule states that if there is no such element, then we return the simplified read on b at the same index, using Λ as the representation.

6.4.4 Refinement: Adding Domain-Based Reasoning

While our equality check performs well for deciding (dis-)equalities between indexes with a same base, it behaves poorly with different bases. So we extend FAS in Figure 6.8 with *domain-based reasoning* abilities. Basically, maps are now equipped with abstract domains over-approximating their sets of (possible) concrete indexes, and the data structure is now a **list of sets of maps**, all maps in a set having different bases but disjoint sets of concrete indexes. When syntactic base/offset equality check is not

$$\frac{i = \beta + \iota \qquad \iota \text{ a constant} \qquad \begin{array}{c}\Theta = \left\{\left\langle \Sigma, \sigma, c, \Sigma^\sharp \right\rangle \mid \sigma \neq \beta \wedge \Sigma^\sharp \sqcap i^\sharp = \bot\right\} \\ \Xi = \left\{\left\langle \Sigma, \sigma, c, \Sigma^\sharp \right\rangle \mid \sigma \neq \beta \wedge \Sigma^\sharp \sqcap i^\sharp \neq \bot\right\}\end{array}}{\left\{\left(\left\langle \Gamma, \beta, b, \Gamma^\sharp \right\rangle \oplus \Theta \uplus \Xi\right) :: \Lambda\right\} a[i] \leftarrow e \left\{\left(\left\langle \Gamma[\iota] \leftarrow e, \beta, b, \Gamma^\sharp \sqcup i^\sharp \right\rangle \oplus \Theta\right) :: \Xi :: \Lambda\right\}} \text{ STORE-HIT}$$

$$\frac{i = \alpha + \iota \qquad \iota \text{ a constant} \qquad \begin{array}{c}\Theta = \left\{\left\langle \Sigma, \sigma, c, \Sigma^\sharp \right\rangle \mid \sigma \neq \alpha \wedge \Sigma^\sharp \sqcap i^\sharp = \bot\right\} \\ \Xi = \left\{\left\langle \Sigma, \sigma, c, \Sigma^\sharp \right\rangle \mid \sigma \neq \alpha \wedge \Sigma^\sharp \sqcap i^\sharp \neq \bot\right\}\end{array}}{\{(\Theta \uplus \Xi) :: \Lambda\} a[i] \leftarrow e \left\{\left(\left\langle \varnothing[\iota] \leftarrow e, \alpha, a, i^\sharp \right\rangle \oplus \Theta\right) :: \Xi :: \Lambda\right\}} \text{ STORE-MISS}$$

$$\frac{\Gamma[\iota] = e \qquad i = \beta + \iota}{\left\{\left(\left\langle \Gamma, \beta, b, \Gamma^\sharp \right\rangle \oplus \Xi\right) :: \Lambda\right\} \vdash a[i] \rightsquigarrow e} \text{ SELECT-HIT}$$

$$\frac{\{\Lambda\} \vdash b[i] \rightsquigarrow e \qquad \Gamma[\iota] = \varnothing \qquad i = \beta + \iota}{\left\{\left(\left\langle \Gamma, \beta, b, \Gamma^\sharp \right\rangle \oplus \Xi\right) :: \Lambda\right\} \vdash a[i] \rightsquigarrow e} \text{ SELECT-MISS}$$

$$\frac{\{\Lambda\} \vdash b[i] \rightsquigarrow e \qquad i = \beta + \iota \qquad \Theta = \left\{\left\langle \Sigma, \sigma, c, \Sigma^\sharp \right\rangle \mid \sigma \neq \beta \wedge \Sigma^\sharp \sqcap i^\sharp = \bot\right\}}{\{\Theta :: \Lambda\} \vdash a[i] \rightsquigarrow e} \text{ SELECT-SKIP}$$

Figure 6.8 – Inference rules for $\cdot[\cdot]$ and $\cdot[\cdot] \leftarrow \cdot$ using domains.

conclusive, domain intersection may be used to prove disequality.

We borrow ideas from Abstract Interpretation [CC77]. Given a concrete domain D, an abstract domain is a complete lattice $\left\langle D^\sharp, \sqsubseteq, \sqcup, \sqcap, \top, \bot \right\rangle$ coming with a monotonic concretization function $\gamma : D^\sharp \mapsto \mathcal{P}(D)$ such that $\gamma(\top) = D$ and $\gamma(\bot) = \varnothing$. An element of an abstract domain is called an abstract value. In the following the concrete domain is the set of array indexes.

The representation is now a list of sets of tuples $\left\langle \Gamma, \beta, b, \Gamma^\sharp \right\rangle$ where Γ, β and b are a map, a base and an array as previously described, and where $\Gamma^\sharp$ is the joined abstract value of indexes in Γ. Given a write at index i, the set at head position in the list is split into: 1) Θ the set of tuples whose map abstract value does not overlap with $i^\sharp$, the abstract value of i, 2) Ξ the set of tuples whose map abstract value overlap with $i^\sharp$, and if it exists, 3) the tuple $\left\langle \Gamma, \beta, b, \Gamma^\sharp \right\rangle$ where β is after normalization the base of i. If this tuple exists, then the STORE-HIT rule applies. We update Γ as previously and its

associated abstract value becomes the join value of $\gamma^\sharp$ and $i^\sharp$. We append first Ξ alone onto the list, and then Θ together with the updated tuple. Else, the STORE-MISS rule applies. Again we first append Ξ alone, then Θ together with a new singleton map, the new base, the written array and the write index abstract value. Finally, SELECT-HIT and SELECT-MISS are similar to previous ones, but we add a new rule SELECT-SKIP. This rule states that, if the read index abstract value do not overlap with maps abstract values in the set at head position, then we drop the head and reiterate on the tail of the list.

Note that if abstract values in these rules are set to $\top$, then Θ is always empty and we get back to the previous inference rules. Also the complexity of reads becomes linear in the list size, as domains can prove disequality at each node of the list. Yet, it is not a problem in practice, as demonstrated by experimental evaluation in Section 6.5.

Domain Propagation So far, we did not explained how abstract values are computed. The literature on abstract domains is plentiful [Sim08]. Nevertheless we present in Figure 6.9 propagation rules for a specific abstract domain, the well-known domain of (multi-)intervals — used in our implementation. Note that operations are performed over bitvectors of a known size N, and that + denotes the wraparound addition. The general difficulty is to find a sweet spot between the potential gain (more checks become conclusive) and the overhead of propagation. As a rule of thumb, *non-relational domains* should be tractable and useful. Especially, combining multi-intervals with congruence (e.g. $x \equiv 5 \bmod 8$) or bit-level information (e.g. the second bit of x is 1) [BHP10] is a good candidate for refining our method at an affordable cost.

6.5 Implementation and Experimental Evaluation

6.5.1 Implementation

In order to evaluate the efficiency of our approach, we implemented FAS (with the different representations presented so far and the abstract domain of multi-intervals) as a preprocessor for SMT formulas belonging to the QF-ABV logic (*quantifier-free formulas over the theory of bitvectors and arrays*, Chapter 3, Section 3.3.2, Section 3.3.3) — as typical choice in software verification. In that setting, all bitvector values and expressions have statically known sizes, arithmetic operations are performed *modulo* and values can "wraparound". For reproducibility purposes source code and benchmarks are available online[2].

[2] http://benjamin.farinier.org/lpar2018/

Let i, j two bitvectors of size N, with $i^\sharp = [m_i, M_i]$, $j^\sharp = [m_j, M_j]$ where $0 \leq m_{i,j} \leq M_{i,j} \leq 2^N$,

$$
\begin{aligned}
c^\sharp &= [\mathrm{c}, \mathrm{c}] && \text{for any constant } c \\
v^\sharp &= [m_i, M_j] && \text{if } i \leq v \leq j \\
(\mathrm{extract}_{l,h}\ i)^\sharp &= [0, 2^{h-l+1} - 1] && \text{if } (M_i \gg l) - (m_i \gg l) \geq 2^{h-l+1} \\
&= [\mathrm{extract}_{l,h}\ (m_i)\ , \mathrm{extract}_{l,h}\ (M_i)] && \text{if } \mathrm{extract}_{l,h}\ (M_i) \geq \mathrm{extract}_{l,h}\ (m_i) \\
&= [0, \mathrm{extract}_{l,h}\ (M_i)] && \text{otherwise} \\
&\quad \sqcup\ [\mathrm{extract}_{l,h}\ (m_i)\ , 2^{h-l+1} - 1] && \\
(i + j)^\sharp &= [m_i + m_j, M_i + M_j] && \text{if } M_i + M_j < 2^N \\
&= [m_i + m_j - 2^N, M_i + M_j - 2^N] && \text{if } m_i + m_j \geq 2^N \\
&= [m_i + m_j - 2^N, 2^N - 1] && \text{otherwise} \\
&\quad \sqcup\ [0, M_i + M_j - 2^N] &&
\end{aligned}
$$

Figure 6.9 – Examples of propagation for intervals. These propagations are extended to multi-intervals by distribution for unary operators and pairwise distribution for binary operators.

The implementation comprises 6 300 lines of OCaml integrated into the Tfml logical formula preprocessing engine [FBBP18], part of the Binsec Symbolic Execution tool [DBT+16]. It comprises all simplifications and optimizations described in Section 6.4, including map lists, base normalization, sub-term sharing and domain propagation (multi-intervals) over bit-vectors. Base normalization and sub-term sharing are applied systematically, and at construction time, thanks to *smart-constructors*: trying to build the term $(\mathtt{esp}_0 - 64) + 48$ will automatically result in building the term $\mathtt{esp}_0 - 16$, the non-simplified version of the term just never exists. Note that our normalization rules (Section 6.4.2) and domain propagators (Section 6.4.4) correctly handle possible arithmetic wraparounds.

An advantage operating as a preprocessor is to be independent of the underlying solver used for formula resolution, and therefore allow us to evaluate the impact of our approach with several of them. A drawback is that we do not have access to various internal components of the solver, like accessing the model under construction, and cannot use them to refine our approach. In the long term, a deeper integration into a solver would be more suitable.

6.5.2 Experimental Setup

We evaluated FAS performances under three criteria : 1) *simplification thoroughness*, measured by the reduction of the number of ROW terms; 2) *simplification impact*, measured by resolution time before and after simplification; 3) *simplification cost*, measured by the total time of simplification.

We devise three sets of experiments corresponding to three different scenarios: mid-sized formulas generated by the SE-tool BINSEC [DBT+16] from real executables programs — typical of test generation and vulnerability finding (Section 6.5.3), very large formulas generated by BINSEC from very long traces — typical of reverse and malware analysis (Section 6.5.4), and formulas taken from the SMT-LIB benchmarks (Section 6.5.5). Regarding experiments over SE-generated formulas, we also consider three variants corresponding to standard concretization / symbolization policies [DBF+16] (cf. Section 6.5.3), as well as different TIMEOUT values. Experiments are carried out on an Intel(R) Xeon(R) CPU E5-2660 v3 @ 2.60GHz. We consider three of the best SMT solvers for the QF-ABVtheory, namely Boolector (Btor) [BB09a], Yices [Dut14] and Z3 [dMB08].

Note that the impact of map lists (with regard to a list-based representation) and sub-term sharing will be evaluated only in Section 6.5.4, as they are interesting only on large enough formulas. Moreover, the map list representation impacts only preprocessing time, not its thoroughness: assuming preprocessing does not time out (and rebase and domains are used), FAS and FAS-list will carry out the same simplifications.

A Note on Problem Encoding As already stated, we consider quantifier-free formulas over the theory of bitvectors and arrays coming from the encoding of low-level software verification problems. Arithmetic operations are performed *modulo* and values can "wraparound". Also, since memory accesses in real hardware are performed at word-level (reading 4 or 8 bytes at once), they are modelled here by successive byte-level reads and writes — allowing to take properly into account misaligned or overlapping accesses. Finally, memory is often modelled as a single logical array of bytes (i.e., bitvector values of size 8), without any *a priori* distinction betweeen stack and heap (this is the case for all examples from BINSEC).

6.5.3 Medium-Size Formulas from SE

We consider here typical formulas coming from Symbolic Execution over executable codes. While mid-sized (max. 3.42 MB, avg. 1.40 MB), these formulas comprise quite

Table 6.1 – 6 590 x 3 medium-size formulas from SE, with TIMEOUT = 1 000 sec.: simplification time (in seconds), number of ROW after simplification, number of TIMEOUT and resolution time (in seconds, without TIMEOUT).

		simpl. time	#TIMEOUT and resolution time						#ROW
			Btor		Yices		Z3		
concrete	default	61	0	163	2	69	0	872	866 155
	FAS	85	0	94	2	68	0	244	1 318
	FAS-itv	111	0	94	2	68	0	224	1 318
interval	default	65	0	2 584	2	465	31	155 992	866 155
	FAS	99	0	2 245	2	487	25	126 806	531 654
	FAS-itv	118	0	755	2	140	14	37 269	205 733
symbolic	default	61	0	6 173	3	1 961	65	305 619	866 155
	FAS	91	0	6 117	3	1 965	66	158 635	531 654
	FAS-itv	111	0	4 767	2	1 108	43	80 569	295 333
total	default	187	0	8 922	7	2 495	96	462 484	2 598 465
	FAS	275	0	8 458	7	2 520	91	285 686	1 064 626
	FAS-itv	340	0	5 616	6	1 317	57	37 573	502 384

long sequences of nested ROW (max. 11 368 ROW, avg. 4 726 ROW) as there is only one initial array (corresponding to the initial memory of the execution, i.e. a flat memory model). More precisely, we consider 6 590 traces generated by BINSEC [DBT+16] from 10 security challenges (e.g. `crackme` such as Manticore or Flare-On) and vulnerability finding problems (e.g. GRUB vulnerability), and from these traces we generate 3 x 6 590 formulas depending on the concretization / symbolization policies used in BINSEC to generate them: **concrete** (all array indexes are set to constant values), **symbolic** (symbolic array indexes), and **interval** (array indexes bound by intervals). We consider two different TIMEOUT: 1 000 seconds (close to SMT-LIB benchmarks setting) and 1 second (typical of program analysis involving a large number of solver calls, e.g. deductive verification or Symbolic Execution).

The whole results are presented in Table 6.1 (TIMEOUT 1 000 sec.) and Table 6.2 (TIMEOUT 1 sec.). Note that resolution time does not include TIMEOUT. Columns FAS and FAS-itv represents respectively our technique (map list, rebase and sharing) potentially improved with domain reasoning based on intervals (FAS-itv). The **default** column represents a minimal preprocessing step consisting of constant propagation and formula pruning, *without any array simplification*.

Table 6.2 – 6 590 x 3 medium-size formulas from SE, with TIMEOUT = 1 sec.

		#TIMEOUT and resolution time					
		Btor		Yices		Z3	
concrete	default	2	93	2	3.12	2	655
	FAS	2	24	2	2.54	2	39
	FAS-itv	2	23	2	2.51	2	40
interval	default	1 230	730	57	184	480	751
	FAS	593	1 213	58	181	483	773
	FAS-itv	52	602	6	66	273	665
symbolic	default	1 947	575	2 771	307	3 497	438
	FAS	1 888	618	2 723	310	3 470	442
	FAS-itv	1 597	647	1 473	528	2 895	504
total	default	3 179	1 399	2 830	494	3 979	1 845
	FAS	2 483	1 856	2 783	495	3 955	1 254
	FAS-itv	1 651	1 273	1 481	597	3 170	1 210

We can see that:

- *Simplification time* is always very low on these examples (340 sec. for 3 x 6 590 formulas, in avg. 0.017 sec. per formula). Moreover, it is also very low with regard to resolution time (taking TIMEOUT into account: Boolector 6%, Yices 4% and Z3 0.3%) and largely compensated by the gains in resolution, but for one case where Boolector performs especially well (concrete formulas: cost of 118% — not compensated by gains in resolution).

- Formula simplification is indeed *thorough*: as a whole, the number of ROW is reduced by a factor 5 (2.5 without interval reasoning). The simplification performs extremely well, as expected, on *concrete* formulas, where almost all ROW instances are solved at preprocessing time. On *interval* formulas, the number of ROW is sliced by a factor 4, and a factor 3 in the case of *full symbolic* formulas.

- The *impact* of the simplification over resolution time (for a 1 000 sec. TIMEOUT) varies greatly from one solver to another, but it is always significant: factor 1.5 for Boolector, factor 1.9 for Yices with one fewer TIMEOUT, up to a factor 3.8 and 32 fewer TIMEOUT for Z3. Especially, on interval formulas FAS with domain reasoning yields a 3.4 (resp. 3.3) speed factor for Boolector (resp. Yices), while Z3 on this category enjoys a 4.1 speedup together with 14 fewer TIMEOUT. Interestingly,

Table 6.3 – GRUB (interval), 753 formulas — Number of TIMEOUT and resolution time (in seconds, without TIMEOUT).

	#TIMEOUT and resolution time					
GRUB	Btor		Yices		Z3	
default	0	508	0	258	0	31 322
FAS	0	505	0	257	1	26 809
FAS-itv	0	123	0	54	0	4 481

Table 6.4 – UNGAR (symbolic), 139 formulas — Number of TIMEOUT and resolution time (in seconds, without TIMEOUT).

	#TIMEOUT and resolution time					
UNGAR	Btor		Yices		Z3	
default	0	359	3	627	12	926
FAS	0	373	3	624	12	1 130
FAS-itv	0	19	2	13	0	569

domain reasoning is useful also in the case of fully symbolic formulas, i.e. with no explicit introduction of domain-based constraints.

Results for a 1 sec. TIMEOUT follows the same trend but they are much more significant (number of TIMEOUT: Boolector -48%, Yices -47% and Z3 -21%), and they become especially dramatic on interval formulas (number of TIMEOUT: Boolector -96%, Yices -90% and Z3 -44%).

Focus on Specific Cases We highlight now a few interesting scenarios where FAS performs very well, especially formulas generated from the GRUB vulnerability we presented in Chapter 2 (Table 6.3, 753 formulas), and formulas representing the inversion of a crypto-like challenge called UNGAR (Table 6.4, 139 formulas). Regarding GRUB, while basic FAS does not really impact resolution time, adding domain-based reasoning does allow a significant improvement — Boolector, Yices and Z3 becoming respectively 4.1x, 4.7x and 7x faster. Regarding UNGAR, again FAS alone does not improve resolution time (for Z3, we even see worse performance), but adding interval reasoning yields dramatic improvement: Boolector becomes 18.8x faster, Yices becomes 48.2x faster (with -1 TIMEOUT) and Z3 does not time out anymore (-12 TIMEOUT).

Conclusion On these middle-size formulas coming from typical SE problems, we can draw the following conclusion:

Speed FAS is extremely efficient and does not yield any noticeable overhead;

Thoroughness Formula simplification is significant — even on fully symbolic formulas, and it becomes (as expected) dramatic on "concrete" formulas;

Impact The impact of FAS varies across solvers and formulas categories, yet it is always positive and it can be dramatic in some settings (low TIMEOUT, interval formulas).

Table 6.5 – 29 x 3 very large formulas from SE, with TIMEOUT = 1 000 sec.: simplification time (in seconds), number of ROW after simplification, number of TIMEOUT and resolution time (in seconds, without TIMEOUT).

		simpl. time	#TIMEOUT and resolution time						#ROW
			Btor		Yices		Z3		
concrete	default	44	10	159	4	1 098	26	3.33	1 120 798
	FAS-list	1 108	8	845	4	198	10	918	456 915
	FAS	196	8	820	4	196	10	922	456 915
	FAS-itv	210	4	654	1	12	4	1 120	0
interval	default	44	12	131	12	596	27	0.19	1 120 798
	FAS-list	222	12	129	12	595	26	236	657 594
	FAS	231	12	129	12	597	26	291	657 594
	FAS-itv	237	12	58	12	28	19	81	651 449
symbolic	default	40	12	1 522	12	1 961	27	0.13	1 120 798
	FAS-list	187	11	1 199	12	2 018	26	486	657 594
	FAS	194	11	1 212	12	2 081	26	481	657 594
	FAS-itv	200	11	1 205	12	2 063	26	416	657 594

6.5.4 Very Large Formulas

We now turn our attention to large formulas (max. 458 MB, avg. 45 MB) involving very long sequences of nested ROW (max. 510 066 ROW, avg. 49 850 ROW), as can be found for example in symbolic deobfuscation. We consider 29 benchmarks taken from a recent article on the topic [SBP18] representing execution traces over (mostly non crypto-) hash functions (e.g. `MD5`, `City`, `Fast`, `Spooky`, etc.) obfuscated by the Tigress tool [CMMN12]. We also consider a trace taken from the ASPack packing tool[3].

Results are presented in Table 6.5, where FAS-list represents our simplification method where the map list is replaced by a normal list — getting an improved version of the standard list-based ROW-simplification (the goal being to evaluate the gain of our new data structure). Again, simplification is significant with a strong impact on the number of time outs and on resolution time, especially in the concrete case and for Z3. Impact in the symbolic case is more mixed but positive (-1 TIMEOUT for Boolector and Z3, no impact for Yices). In term of size, FAS reduces formulas to max. 86.49MB, avg. 6.98MB, and FAS-itv to max. 86.45MB, avg. 6.17MB. If sub-term sharing is disabled, formulas size jumps to max. 591.99MB, avg. 14.95MB for FAS and max. 591.71MB, avg. 16.35MB for

[3] http://www.aspack.com/

Table 6.6 – ASPack formula, without TIMEOUT.

ASPack	simpl. time	resolution time			#ROW
		Btor	Yices	Z3	
default	15 sec.	≈ 24h	69 sec.	2h36	360 991
FAS-list	53 min.	9.7 sec.	3.4 sec.	183 sec.	0
FAS	61 sec.	9.7 sec.	3.4 sec.	183 sec.	0
FAS-ITV	63 sec.	9.8 sec.	3.4 sec.	182 sec.	0

FAS-ITV. Regarding simplification time, FAS-list suffers from scalability issues on these formulas (5x slower than FAS).

The ASPack Example We now turn our attention to the formula generated from a trace of a program protected by ASPack (96 MB and 363 594 ROW, *concrete* mode). Solving the formula is highly challenging: while Yices succeeds in a decent amount of time (69 seconds), Z3 terminates in 2h36min while Boolector needs 24h. Table 6.6 presents our results on this particular example. FAS performs extremely well (Table 6.6), turning resolution time from hours to a few seconds (Boolector) or minutes (Z3). Yices also benefits from it. Especially, all ROW instances are simplified away. FAS and FAS-ITV reduce the ASPack formula size to 3.81MB, while it jumps to 443.54MB when sub-term sharing is disabled. Interestingly, this example clearly highlights the scalability of FAS with regard to a standard list-based approach, passing roughly from 1h (list) to 1 minute (FAS). Figure 6.10 proposes a detailed view of the performance and impact of the standard list-based simplification method (Boolector only), depending on the bound for backward reasoning (the standard method has no bound). For comparison, the two horizontal lines represent simplification and resolution time with FAS. We can see that bounding the list-based reasoning has no tangible effect here, as we need at least a 3 000 seconds (50 minutes) simplification time to get a resolution time under 3 000 seconds.

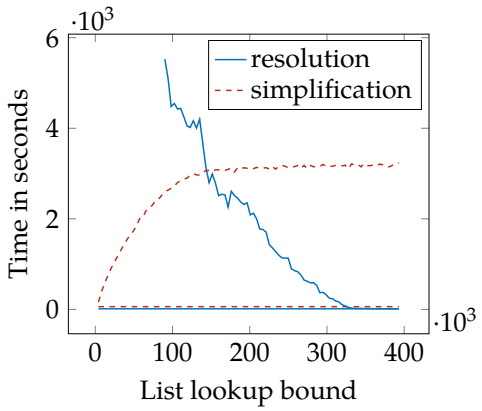


Figure 6.10 – Boolector on ASPack.

Conclusion Once again FAS appears to be fast and to have a significant impact on resolution time, especially in the concrete case where the difference can be from several

Table 6.7 – 15 016 formulas from SMT-LIB benchmarks, with TIMEOUT = 1.000 sec.: simplification time (in seconds), number of ROW after simplification, number of TIMEOUT and resolution time (in seconds, without TIMEOUT).

SMT-LIB	simpl. time	#TIMEOUT and resolution time						#ROW
		Btor		Yices		Z3		
default	87	59	20 126	151	28 156	158	41 925	548 176
FAS	378	54	19 922	148	26 657	147	43 090	469 815
FAS-itv	378	55	19 843	146	28 703	149	40 873	469 567

hours to a few seconds (total resolution + simplification: a few minutes). Moreover, it appears clearly that on very long traces FAS scales much better than the standard list-based ROW-simplification method.

6.5.5 SMT-LIB Formulas

We consider now the impact of FAS on formulas taken from the SMT-LIB benchmarks. These formulas are notably different from the ones considered in the two previous experiments: while most of them do come from verification problems, they may involve complex Boolean structure (rather than "mostly conjunctive" formulas) and they do not necessarily exhibit very deep chains of ROW. These kinds of formulas are not our primary objective, yet we seek to evaluate how our technique performs on a "bad case". We evaluate FAS on all the 15 016 SMT-LIB formulas from QF-ABVtheory. TIMEOUT is set to 1 000 seconds. Results are reported in Table 6.7. Note that, again, resolution time does not include TIMEOUT.

Conclusion FAS is again very efficient on these formulas (avg. 0.025 sec. per formula), and reduces the number of ROW by -14%. Yet the impact of simplifications, while slight, is clearly positive on both TIMEOUT (Boolector -8%, Yices -2% and Z3 -7%) and resolution time (for Yices, only when taking TIMEOUT time into account). Such gains are not anecdotal as the best SMT solvers are highly tuned for SMT-LIB. Since the number of TIMEOUT is the main metric for SMT-LIB, Boolector with FAS would have won the last edition for QF-ABVtheory. Finally, domain reasoning does not add anything here (but for Yices) — either the benchmark formulas do not exhibit such interval constraints, or our propagation mechanism is too crude to take advantage of it.

6.5.6 Conclusion

Our experiments demonstrate that our approach is *efficient* (the cost is almost always negligible with regard to resolution time) and *scalable* (compared to the list-based method). The simplification is *thorough*, removing a large fraction of ROW. The *impact is always positive* (both in resolution time and number of time outs), and it is *dramatic for some key usage scenarios* such as SE-like formulas with small TIMEOUT or very large size.

Finally, we can note that domain reasoning is usually helpful (though, not on SMT-LIB formulas) and that it shows a powerful synergy with the "interval C/S policy" in SE — yielding a new interesting sweet spot between tractability and genericity of reasoning.

6.6 Related Works

Decision Procedures for the Theory of Arrays Surprisingly, there have been relatively few works on the efficient handling of the (basic) theory of arrays. Standard symbolic approaches for pure arrays complement symbolic read-over-write preprocessing [GD07, BNO+08, BG12] with enumeration on (dis-)equalities, yielding a potentially huge search space. New array lemmas can be added on-demand or incrementally discovered through an abstraction-refinement scheme [BB09b]. Another possibility is to reduce the theory of arrays to the theory of equality by systematic "inlining" of the array axioms to remove all $\cdot[\cdot] \leftarrow \cdot$ operators, at the price of introducing many case-splits The encoding can be eager [KS08] or lazy [BB09b]. Our method generalizes previous preprocessings [GD07, BG12] and is complementary to complete resolution methods [BB09b, KS08]. Note also that our approach could benefit from being integrated directly within such a complete resolution method, allowing incremental simplification all along the resolution process.

Decision procedures have also been developed for expressive extensions of the array theory, such as arrays with extensionality (i.e. equality over whole arrays) or the array property fragment [BMS06], which enables limited forms of quantification over indexes and arithmetic constraints. These extensions aim at increasing expressiveness and they do not focus so much on practical efficiency. Our method can also be applied to these settings (as ROW are still a crucial issue), even though it will not cover all difficulties of these extensions.

Optimized Handling of Arrays Inside Tools Many verification and program analysis tools and techniques ultimately rely on solving logical formulas involving the theory of

arrays. Since the common practice is to re-use existing (SMT) solvers, these approaches suffer from the limitations of the current solvers over arrays. As a mitigation, some of these tools take into account knowledge from the application domain in order to generate relevant (but usually not equivalent) and simpler formulas [PMZC17, FLP16] — see also the specific case of SE over concrete indexes discussed in Section 6.3. Our method is complementary to these approaches as it operates on arbitrary formulas and the simplification keeps logical equivalence.

6.7 Conclusion

The theory of arrays has a central place in software verification due to its ability to model memory or data structures. Yet, this theory is known to be hard to solve because of *read-over-write* terms (ROW), especially in the case of very large formulas coming from unrolling-based verification methods. We have presented FAS, an original simplification method for the theory of arrays geared at eliminating ROW, based on a new dedicated data structure together with original simplifications and low-cost reasoning. The technique is efficient, scalable and it yields significant simplification. The impact on formula resolution is always positive, and it can be dramatic on some specific classes of problems of interest, like very long formulas coming from a binary-level Symbolic Execution as the one we present in Chapter 7. These advantages have been experimentally proven both on realistic formulas coming from Symbolic Execution and on SMT-LIB formulas.

Future work includes a deeper integration inside a dedicated array solver in order to benefit from more simplification opportunities along the resolution process, as well as exploring the interest of adding more expressive domain reasoning.

Bibliography

[BB09a] Robert Brummayer and Armin Biere. Boolector: An Efficient SMT Solver for Bit-Vectors and Arrays. In *Tools and Algorithms for the Construction and Analysis of Systems, 15th International Conference, TACAS 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings*, pages 174–177, 2009.

[BB09b] Robert Brummayer and Armin Biere. Lemmas on demand for the extensional theory of arrays. *JSAT*, 6(1-3):165–201, 2009.

[BDM17] Sébastien Bardin, Robin David, and Jean-Yves Marion. Backward-bounded DSE: targeting infeasibility questions on obfuscated codes. In *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*, pages 633–651. IEEE, 2017.

[BG12] Sébastien Bardin and Arnaud Gotlieb. FDCC: A combined approach for solving constraints over finite domains and arrays. In *Integration of AI and OR Techniques in Contraint Programming for Combinatorial Optimzation Problems - 9th International Conference, CPAIOR 2012, Nantes, France, May 28 - June1, 2012. Proceedings*, pages 17–33, 2012.

[BHP10] Sébastien Bardin, Philippe Herrmann, and Florian Perroud. An alternative to sat-based approaches for bit-vectors. In *Tools and Algorithms for the Construction and Analysis of Systems, 16th International Conference, TACAS 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings*, pages 84–98, 2010.

[BM07] Aaron R. Bradley and Zohar Manna. *The Calculus of Computation: Decision Procedures with Applications to Verification (Chapters 5, 6 and 12)*. Springer, 2007.

[BM18] Sylvie Boldo and Nicolas Magaud, editors. *Journées Francophones des Langages Applicatifs, JFLA 2018, Banyuls-sur-Mer, France, January 24-27, 2018*, 2018.

[BMS06] Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. What's decidable about arrays? In *Verification, Model Checking, and Abstract Interpretation, 7th International Conference, VMCAI 2006, Charleston, SC, USA, January 8-10, 2006, Proceedings*, pages 427–442, 2006.

[BNO+08] Miquel Bofill, Robert Nieuwenhuis, Albert Oliveras, Enric Rodríguez-Carbonell, and Albert Rubio. A write-based solver for SAT modulo the theory of arrays. In *Formal Methods in Computer-Aided Design, FMCAD 2008, Portland, Oregon, USA, 17-20 November 2008*, pages 1–8, 2008.

[BT18] Clark W. Barrett and Cesare Tinelli. Satisfiability modulo theories. In *Handbook of Model Checking.*, pages 305–343. 2018.

[CC77] Patrick Cousot and Radia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *4th ACM Symposium on Principles of Programming Languages (POPL)*. ACM, 1977.

[CKL04] Edmund M. Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In *TACAS, Barcelona, Spain, March 29 - April 2, 2004*, pages 168–176, 2004.

[CMMN12] Christian S. Collberg, Sam Martin, Jonathan Myers, and Jasvir Nagra. Distributed application tamper detection via continuous software updates. In *28th Annual Computer Security Applications Conference, ACSAC 2012, Orlando, FL, USA, 3-7 December 2012*, pages 319–328, 2012.

[CS13] Cristian Cadar and Koushik Sen. Symbolic execution for software testing: three decades later. *Commun. ACM*, 56(2):82–90, 2013.

[DBF+16] Robin David, Sébastien Bardin, Josselin Feist, Laurent Mounier, Marie-Laure Potet, Thanh Dinh Ta, and Jean-Yves Marion. Specification of concretization and symbolization policies in symbolic execution. In *ISSTA, Saarbrücken, Germany, July 18-20, 2016*, pages 36–46, 2016.

[DBT+16] Robin David, Sébastien Bardin, Thanh Dinh Ta, Laurent Mounier, Josselin Feist, Marie-Laure Potet, and Jean-Yves Marion. BINSEC/SE: A Dynamic Symbolic Execution Toolkit for Binary-Level Analysis. In *IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering, SANER 2016, Osaka, Japan, March 14-18, 2016 - Volume 1*, pages 653–656, 2016.

[dMB08] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, pages 337–340, 2008.

[DS78] Peter J. Downey and Ravi Sethi. Assignment commands with array references. *J.ACM*, 25(4):652–666, 1978.

[Dut14] Bruno Dutertre. Yices 2.2. In *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, pages 737–744, 2014.

[FBBP18] Benjamin Farinier, Sébastien Bardin, Richard Bonichon, and Marie-Laure Potet. Model generation for quantified formulas: A taint-based approach. In *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part II*, pages 294–313, 2018.

[FDBL18] Benjamin Farinier, Robin David, Sébastien Bardin, and Matthieu Lemerre. Arrays made simpler: An efficient, scalable and thorough preprocessing. In *LPAR-22. 22nd International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Awassa, Ethiopia, 16-21 November 2018*, pages 363–380, 2018.

[FLP16] Aymeric Fromherz, Kasper Søe Luckow, and Corina S. Pasareanu. Symbolic arrays in symbolic pathfinder. *ACM SIGSOFT Software Engineering Notes*, 41(6):1–5, 2016.

[GD07] Vijay Ganesh and David L. Dill. A decision procedure for bit-vectors and arrays. In *Computer Aided Verification, 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007, Proceedings*, pages 519–531, 2007.

[GKS05] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: directed automated random testing. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12-15, 2005*, pages 213–223, 2005.

[KS08] Daniel Kroening and Ofer Strichman. *Decision Procedures - An Algorithmic Point of View*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2008.

[PMZC17] David Mitchel Perry, Andrea Mattavelli, Xiangyu Zhang, and Cristian Cadar. Accelerating array constraints in symbolic execution. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, Santa Barbara, CA, USA, July 10 - 14, 2017*, pages 68–78, 2017.

[Rus05] John M. Rushby. Automated test generation and verified software. In *Verified Software: Theories, Tools, Experiments, First IFIP TC 2/WG 2.3 Conference, VSTTE 2005, Zurich, Switzerland, October 10-13, 2005, Revised Selected Papers and Discussions*, pages 161–172, 2005.

[SBP18] Jonathan Salwan, Sébastien Bardin, and Marie-Laure Potet. Symbolic deobfuscation: From virtualized code back to the original. In *Detection of Intrusions and Malware, and Vulnerability Assessment - 15th International Conference, DIMVA 2018, Saclay, France, June 28-29, 2018, Proceedings*, pages 372–392. Springer, 2018.

[Sim08] Axel Simon. *Value-Range Analysis of C Programs: Towards Proving the Absence of Buffer Overflow Vulnerabilities*. Springer, 2008.

[YJWD15] Babak Yadegari, Brian Johannesmeyer, Ben Whitely, and Saumya Debray. A generic approach to automatic deobfuscation of executable code. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*, pages 674–691. IEEE, 2015.

Chapter 7

Get Rid of False Positives with Robust Symbolic Execution

As explained in Chapter 1, Symbolic Execution is an under-approximating formal verification technique which was proven successful in bug finding, particularly for its absence of false positives: a reported bug is a real bug. However, if this property is true when the user controls all the program inputs, things get more complicated when some inputs are not controllable, typically the environment. In that situation, Symbolic Execution becomes fragile in the sense that it can produce false positives. As shown in Chapter 2, this is especially the case in vulnerabilities analysis, where exploits must be reproducible. In this chapter, we show first results on how to move from the classic reachability problem to the novel *robust reachability* problem by the use of quantifiers. In order to minimize the impact on resolution time, these quantifiers are then eliminated using the taint-based approach presented in Chapter 5, and obtained formulas are simplified using the technique dedicated to array terms presented in Chapter 6. This results in an efficient Symbolic Execution *robust* with regard to the environment, and correct as an under-approximating verification technique, for it is really free of false positives. *The content of this chapter is a preliminary work, and was presented in [MD19].*

7.1 Introduction

Context Program verification is an undeniable success when applied to critical software. Substantial works were devoted for several years to adapt it to non-critical software. In particular, Symbolic Execution [CS13] we presented in Chapter 4 seems well suited to this effort. Symbolic Execution aims to explore paths of a program in

order to find bugs. Each path comes with an entry obtained by symbolic reasoning and Satisfiability Modulo Theories (SMT) [BT18] which makes the execution follow that path. This method has been a huge success in recent years [GKS05] for its ability to handle complex codes and its absence of false positives: every reported bug is real. Symbolic Execution is thus part of under-approximating software verification techniques.

Problem However, as shown in Chapter 2, false positives exist in practice [CDE08, CKC11]. They come from abstractions of certain pieces of code that are absent or too complicated to handle, for example embedded assembly language in C-level analysis, or for example imperfect models of the initial state or of the environment in binary-level analysis. This problem is known in the Symbolic Execution community, but not really studied nor quantified. It is rather seen as a fatality, not necessarily annoying provided the analysis finds enough bugs. However, it is a serious problem for the user, who can no longer trust calculated coverage rates in a test oriented scenario, or has to spend time analyzing a bug which was reported, but unconfirmed by its associated entry in a security oriented scenario. The problem is particularly salient for binary-level analysis, because user inputs and interactions with the environment are not clearly distinguishable, and therefore are not easy to specify a priori. In practice, ad hoc solutions may reduce the number of false positives, but do not guarantee their absence, and even sometimes introduce new ones or completely overload underlying constraint solvers.

Proposal We aim to develop a Robust Symbolic Execution, which is really exempt of false positives. To this end we define the robust reachability problem, a new framework in which inputs are partitioned between those controllable by the user and those uncontrollable, like the environment. A solution is said to be robust if it reaches the desired goal regardless of uncontrollable values. This framework makes it possible to correctly handle annoying cases mentioned above.

Contributions In Chapter 2, we shown intuitively the interest of Robust Symbolic Execution. This chapter makes the following contributions:

- We formally define in Section 7.2 the framework of robust reachability, and propose a revision of Symbolic Execution, called Robust Symbolic Execution, which aims to solve the robust reachability problem and eliminate false positives.

- Then in Section 7.3 we describe a first implementation of Robust Symbolic Execution in the binary-level Symbolic Execution tool Binsec [DB15, DBT+16], and report some encouraging initial experimental results.

7.2 Robust Symbolic Execution

In this section we describe Robust Symbolic Execution, a Symbolic Execution which is exempt of false positive. In order to achieve this goal, Robust Symbolic Execution distinguish inputs according to their controllability. By "controllability" we designate the ability of an external controller (the user, or an attacker) to choose, or at least influence, the value of a controllable program component (an input variable for example, or the result of the execution of a function controlled by the user).

7.2.1 Robust Reachability

As stated in Chapter 4, Section 4.4.1, verifying if a property holds on a program can be reduced to a reachability problem in an instrumented version of the program, which makes the notion of reachability central for Symbolic Execution. Thus, after having recalled the classic notion of reachability, we define the notion of robust reachability.

Definition 7.1 (Classic Reachability)**.** Let P be a program taking (a, x) as inputs, and let l be a location in the code of P. We said that l is reachable if there exists a_0 and x_0 such that the execution of P (a_0, x_0) reaches location l.

Note that all code locations are not necessarily reachable: there may exist code locations which are not reachable by any entry. If having large portions of non-reachable code is often considered improper — we talk about dead-code locations, it may sometimes be desired, as for example for an **abort** instruction after an assertion check.

Definition 7.2 (Robust Reachability)**.** Let P be a program taking (a, x) as inputs, where a is controllable by the user and where x is uncontrollable, and let l be a location in the code of P. We said that l is robustly reachable if there exists a controllable input a_0 such that, for all uncontrollable input x_0, the execution of P (a_0, x_0) reaches location l.

By taking user inputs for controllable inputs, and the environment for uncontrollable inputs, a location is classically reachable if it is possible to find a user input for which the execution reaches this location for a suitable environment configuration; while a location is robustly reachable if it is possible to find a user input for which the execution reaches this location regardless of the environment configuration.

7.2.2 Robust Symbolic Execution

When classic Symbolic Execution aims to solve the classic reachability problem, Robust Symbolic Execution aims to solve the robust reachability problem. The computed path constraint must now ensure the robust reachability of the targeted path successive locations, and not just their classic reachability. Computed such a robust path constraint is easy: it suffices to compute the classic path constraint, and then to existentially quantify controllable variables and universally quantify uncontrollable variables, as shown in Algorithm 7.1. Doing so requires to distinguish between controllable and uncontrollable inputs, which is discussed later.

Algorithm 7.1: Robust Symbolic Execution of a program P.

forall PC *obtained by classic Symbolic Execution of* P **do**
  Let $\mathcal{I}_\exists \triangleq \{u_1, \ldots, u_{n_\exists}\}$ the set of controllable variables appearing in PC
  Let $\mathcal{I}_\forall \triangleq \{v_1, \ldots, v_{n_\forall}\}$ the set of uncontrollable variables appearing in PC
  if $\{\mathcal{M} \models \exists u_1, \ldots, u_{n_\exists}.\forall v_1, \ldots, v_{n_\forall}.\text{PC}\} \neq \varnothing$ **then**
    return $\{\mathcal{M} \models \exists u_1, \ldots, u_{n_\exists}.\forall v_1, \ldots, v_{n_\forall}.\text{PC}\}$
  else continue

Finally we obtained a path constraint in the following form: $\exists a.\forall x.\text{PC}\,(a, x)$. Note that all existential quantifications appear before universal quantifications, regardless the order in which variables are introduced during the Symbolic Execution. Indeed, existential variables can be chosen in function of preceding universal variables, which means from the program point of view being able to choose some controllable inputs in function of some uncontrollable inputs. It may be possible in specific circumstances when values of some uncontrollable inputs are leaked through program outputs, but it is not possible in general.

The question is now how to solve such quantified path constraints, as quantifications make many theories undecidable — this is the case for the arrays theory. Fortunately, we can reuse some recent extensions of SMT solvers like for example the one for quantified bit vectors, or our taint-based quantifier elimination [FBBP18] we presented in Chapter 5, but also our array terms simplification [FDBL18] we presented in Chapter 6.

7.2.3 Controllable and Uncontrollable Inputs

Compared to classic Symbolic Execution, robust Symbolic Execution requires the user to distinguish between controllable and uncontrollable inputs. For LOW program, this

distinction is clear: the only uncontrollable component is the initial memory, inputs retrieve from **get** instructions are all controllable by the user. But for most programs, distinguishing between controllable and uncontrollable inputs is not so easy. For example it requires for C programs a correct semantics of external calls, and a precise knowledge of the initial memory layout for binary programs. Moreover this distinction is crucial: If the user declares inputs which are uncontrollable in the real system as controllable inputs, the robust reachability problem will be ill-posed and Symbolic Execution will return false positives with regard to the initial "real" problem.

However we can notice that declaring too many entries as uncontrollable still result in a Robust Symbolic Execution exempt of false positives, but at the price of a loss in generality, i.e. at the price of a potential increase in the number of false negatives. A robust way to distinguish controllable and uncontrollable inputs is therefore to consider all inputs uncontrollable by default. An input is consider as controllable only if explicitly specified as such by the user, which is exactly the opposite of that classic Symbolic Execution do.

Semi-Automatic Incremental Specification Although we consider all inputs as uncontrollable by default, the user still has the tedious responsibility to explicitly specify which inputs are controllable. In order to assist the user in this task, we propose in Algorithm 7.2 a semi-automatic incremental specification procedure for inputs which require the user to specify, only when necessary, if an input is controllable. This specification procedure is semi-automatic in the sense that it is automatically driven by Symbolic Execution but requests times to times the user's intervention, and is incremental in the sense that it requires to specify if an input is controllable only when necessary, when the input is used, one input after the others.

Let $\mathcal{T}$ be a set of targets, i.e. a set of code program location we want to achieve, and let $\mathcal{I}$ be the set of program inputs. $\mathcal{T}$ can be either provided by the user, or by default be the set of all code program locations. We first split into three $\mathcal{I} \triangleq \mathcal{I}_\exists \bigcup \mathcal{I}_\forall \bigcup \mathcal{I}_?$ where $\mathcal{I}_\exists$ is the (potentially empty) set of inputs we already known to be controllable, where $\mathcal{I}_\forall$ is the (potentially empty) set of inputs we already known to be uncontrollable, and where $\mathcal{I}_?$ is the set of undetermined inputs considered as uncontrollable by default. The idea is to perform Robust Symbolic Execution by existentially quantifying over controllable variables, universally quantifying over uncontrollable and undetermined variables, but to let the solver to consider some undetermined variables as controllable:

1) Let N be the number of undetermined variables the solver can choose to consider as controllable.

Algorithm 7.2: Semi-automatic incremental specification procedure for inputs.

Let $\mathcal{T}$ a set of targeted program locations
Let $\mathcal{I}_\exists \triangleq \{u_1, \ldots, u_{n_\exists}\}$ the set of known to be controllable program inputs
Let $\mathcal{I}_\forall \triangleq \{v_1, \ldots, v_{n_\forall}\}$ the set of known to be uncontrollable program inputs
Let $\mathcal{I}_? \triangleq \{w_1, \ldots, w_{n_?}\}$ the set of undetermined program inputs
Let $N \triangleq 1$ the number of undetermined inputs to consider

while $\mathcal{I}_? \neq \varnothing$ *and there exists some unachieved targets in* $\mathcal{T}$ **do**
- Let $\iota_1, \ldots, \iota_N$ some fresh "index" symbols
- Let $v_1, \ldots, v_N$ some fresh "value" symbols
- Run Symbolic Execution using the following premise:

$$\exists u_1, \ldots, u_{n_\exists}. \forall v_1, \ldots, v_{n_\forall}. \forall w_1, \ldots, w_{n_?}.$$
$$\bigwedge_{1 \leq i \leq N} \left(\bigvee_{1 \leq j \leq n_?} \left(\iota_i = j \wedge v_i = w_j \right) \right)$$
$$\Rightarrow$$
$$(0 < \iota_1 < \cdots < \iota_{n_?} \leq n_?) \bigwedge \text{PC}$$

- **if** *a new target* $t \in \mathcal{T}$ *is achieved with* $\mathcal{M}$ *a solution of the preceding constraint* **then**
  - Mark t as achieved
  - **for** $1 \leq i \leq N$ **do**
    - Let $\mathtt{i}$ be the valuation of ι_i in $\mathcal{M}$
    - Ask to the user if $w_\mathtt{i}$ is controllable
    - **if** $w_\mathtt{i}$ *is controllable* **then** $\mathcal{I}_\exists \triangleq \mathcal{I}_\exists \bigcup \{w_\mathtt{i}\}$
    - **else** $\mathcal{I}_\forall \triangleq \mathcal{I}_\forall \bigcup \{w_\mathtt{i}\}$ and mark t as unachieved
    - $\mathcal{I}_? \triangleq \mathcal{I}_? \setminus \{w_\mathtt{i}\}$
- **else**
  - **if** $N < n_?$ **then** $N \triangleq N + 1$
  - **else abort**

(a) In order to encode the fact that the solver can select N undetermined variables and choose their values, we introduce N fresh "index" symbols $\iota_1, \ldots, \iota_N$ and N fresh "value" symbols $v_1, \ldots, v_N$. The "index" symbol ι_i models the index of the ith undetermined variable chosen to be consider as controllable, and the "value" symbol v_i models the value given to this variable.

(b) Then the choice is encoded by the premise $\bigwedge_{1 \leq i \leq N} \left(\bigvee_{1 \leq j \leq n_?} \left(\iota_i = j \wedge v_i = w_j \right) \right)$ of an implication whose conclusion is the path constraint PC. We also add the constraint $(0 < \iota_1 < \cdots < \iota_{n_?} \leq n_?)$ to force indexes to belong to the correct range and to be pairwise distinct.

2) If by running Symbolic Execution with this premise we achieved a new target:

(a) We first mark the target as presumably achieved;

(b) Then we look into the solution provided by the solver for values of $\iota_1, \ldots, \iota_N$, and for each valuation $\mathtt{i}$ of ι_i, we ask to the user if input $w_\mathtt{i}$ is controllable:

- If the user answers yes, we move variable $w_\mathtt{i}$ from the set of undetermined inputs to the set of controllable inputs;
- If the user answers no, we move variable $w_\mathtt{i}$ from the set of undetermined inputs to the set of uncontrollable inputs. Moreover in this case, the presumably achieved target was not and is marked back as unachieved, as Symbolic Execution used as controllable a variable which is uncontrollable.

3) If we fail to achieved a new target, we increase N the number of undetermined variables to consider until it exceeds the number of undetermined variables.

Finally, note that it is possible to have even more fine-grained inputs specifications for containers datatype like arrays or bit-vectors. Instead of marking the whole container as controllable, we can choose to mark as controllable only portions of it. To this end, we use the same technique as presented before, but this time the "index" symbol models an index in the container, and the "value" symbol models the value stored in the container at this index. The disjunction $\vee_{1 \leq j \leq n_?} \left(\iota_i = j \wedge v_i = w_j \right)$ in the premise can also be simplified using datatype specific operators, and becomes for example for b a bit-vector $b\,[\iota_i, \iota_i] = v_i$ or for example for a an array $a[\iota_i] = v_i$.

7.3 Implementation and Experimental Evaluation

7.3.1 Implementation

The various steps required for Robust Symbolic Execution are implemented in the Binsec Symbolic Execution tool [DB15, DBT+16]. By default, Binsec in robust mode considers all inputs as uncontrollable. A configuration file is used to specify which inputs are controllable by the user.

Tfml is the logical formulas preprocessing engine of Binsec. On the quantifier elimination side, Tfml implements the generic taint-based procedure presented in Chapter 5, in particular its specialization on the arrays theory. For each array in the initial formula, Tfml introduces a shadow array which is used to follow initial array cells which depend on universally quantified variables as explained in Section 5.6.1. It avoids the introduction of a large number of **if** · **then** · **else** · and allows constraints introduced by our taint-based quantifier elimination to benefit from our simplifications dedicated to array terms. On the simplification side, Tfml systematically applies to terms rewriting rules presented in Chapter 6, Section 6.4.2 at construction time, thanks to smart constructors. Moreover, Tfml applies our treatment for read-over-write once before quantifiers elimination in order to facilitate pointers reasoning, and once after to simplify introduced constraints.

7.3.2 Experimental Evaluation

We evaluate the impact of our approach according to two criteria: 1) the number of true positives, which must be degraded as little as possible; and 2) the number of false positives, which must be reduced to zero. To this end, we consider a set of `crackme` challenges coming from the Binsec test suit whose goal is to find an input for which the program reveals its secret. This kind of program is interesting because it is possible to validate a solution simply by replaying the challenge with. We compare our Robust Symbolic Execution with quantifier elimination and array terms simplification to classic Symbolic Execution with array terms simplification, and to a Robust Symbolic Execution with array terms simplification but without quantifier elimination.

Experiments are performed on an Intel Core i7-4712HQ @ 2.30GHz processor with 16GB of RAM, and results are presented in Table 7.1. Classic Symbolic Execution finds up to 12 correct solutions for these challenges depending on the underlying solver. However, it also produces 9 to 12 false positives which are indistinctly mixed with these solutions and which do not solve corresponding challenges. Robust Symbolic Execution without quantifier elimination is, as intended, free of false positives. Nevertheless the

Table 7.1 – Number of true and false positives for classic Symbolic Execution, Robust Symbolic Execution without quantifier elimination, and Robust Symbolic Execution with quantifier elimination. Robust Symbolic Execution without quantifier elimination is not applicable with Boolector and Yices as they are quantifier-free solvers.

	classic SE		
	true positive	false positive	UNKNOWN
Btor	**12**	11	1
CVC4	7	9	8
Yices	7	11	6
Z3	**12**	12	0

	robust SE		
	true positive	false positive	UNKNOWN
Btor	N/A	N/A	N/A
CVC4	5	**0**	19
Yices	N/A	N/A	N/A
Z3	7	**0**	17

	robust SE + elim.		
	true positive	false positive	UNKNOWN
Btor	**12**	**0**	12
CVC4	7	**0**	17
Yices	7	**0**	17
Z3	**12**	**0**	12

number of true positives is strongly impacted, falling to 7 for the best solver. Finally our Robust Symbolic Execution with quantifier elimination manages to find back all the true positives that classic Symbolic Execution found while remaining free of false positives.

Application to Vulnerability Analysis We also evaluate our Robust Symbolic Execution to the case study inspired by the security vulnerability Back to 28 we presented in Chapter 2. Once again, classic Symbolic Execution returns a non reproducible solution, while underlying solvers fail to solve formulas generated by Robust Symbolic Execution without quantifier elimination. Our Robust Symbolic Execution with our quantifier elimination finds a reproducible solution in 80 seconds, and even in 30 seconds when adding our array terms simplifications.

7.4 Related Works and Conclusion

Related Works As said before, there do not exist to the best of our knowledge other systematic and correct approaches to the robust reachability problem. Current Symbolic Execution tools only solve the non-robust classic reachability problem, and in practice users have to face the false positives issue using ad hoc approaches. For example, for the ill-defined initial state problem which is particularly crucial for bit-level analysis, we find the following mitigations:

- Initializing the memory to an arbitrary value, for example 0: it removes some false positives but also add others, for example in the case where 0 is a solution of the path constraint but does not correspond to what the underlying OS / architecture really is.

- Concretizing the memory with values observed at runtime: once again it removes some false positives — some values are actually constant from one execution to another, but also add others — other values are nonconstant from one execution to another; moreover, the induced concretization constraint will often be too large to be handled by solvers.

Conclusion The Robust Symbolic Execution framework makes it possible to remedy definitively to the false positives problem, provided that: 1) solvers manage to handle generated quantified formulas; and 2) the user correctly specify which inputs are controllable and which inputs are not. Our recent work [FBBP18, FDBL18] we presented in Chapter 5 and Chapter 6 shows how a preprocessing approach allow to reuse quantifier-free solvers to deal with point 1), while we proposed in Algorithm 7.2 a semi-automatic incremental specification procedure to ease point 1). A proof of concept is implemented in the Symbolic Execution tool Binsec.

Bibliography

[BT18] Clark W. Barrett and Cesare Tinelli. Satisfiability modulo theories. In *Handbook of Model Checking.*, pages 305–343. 2018.

[CDE08] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In *8th USENIX Symposium on Operating Systems Design and Implementation,*

OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings, pages 209–224, 2008.

[CKC11] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. S2E: a platform for in-vivo multi-path analysis of software systems. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2011, Newport Beach, CA, USA, March 5-11, 2011*, pages 265–278, 2011.

[CS13] Cristian Cadar and Koushik Sen. Symbolic execution for software testing: three decades later. *Commun. ACM*, 56(2):82–90, 2013.

[DB15] Adel Djoudi and Sébastien Bardin. BINSEC: binary code analysis with low-level regions. In *Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, pages 212–217, 2015.

[DBT+16] Robin David, Sébastien Bardin, Thanh Dinh Ta, Laurent Mounier, Josselin Feist, Marie-Laure Potet, and Jean-Yves Marion. BINSEC/SE: A Dynamic Symbolic Execution Toolkit for Binary-Level Analysis. In *IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering, SANER 2016, Osaka, Japan, March 14-18, 2016 - Volume 1*, pages 653–656, 2016.

[FBBP18] Benjamin Farinier, Sébastien Bardin, Richard Bonichon, and Marie-Laure Potet. Model generation for quantified formulas: A taint-based approach. In *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part II*, pages 294–313, 2018.

[FDBL18] Benjamin Farinier, Robin David, Sébastien Bardin, and Matthieu Lemerre. Arrays made simpler: An efficient, scalable and thorough preprocessing. In *LPAR-22. 22nd International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Awassa, Ethiopia, 16-21 November 2018*, pages 363–380, 2018.

[GKS05] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: directed automated random testing. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12-15, 2005*, pages 213–223, 2005.

[MD19] Nicolas Magaud and Zaynah Dargaye, editors. *Journées Francophones des Langages Applicatifs, JFLA 2019, Les Rousses, France, January 30-February 2, 2019*, 2019.

Part IV

Conclusion

Chapter 8

Conclusion

Symbolic Execution is an automated software verification technique which has proven to be very effective for bug finding. As an under-approximating verification technique, Symbolic Execution should be exempt of false positives. Because of its success, the question is now how to use Symbolic Execution in other contexts than bug finding, for example in vulnerability analysis. But applying Symbolic Execution to vulnerability analysis fundamentally differs from bug finding; and in this new context, false positives appears. This thesis aims to solve this false positives issue, by the mean of several works in the field of decision procedure and software analysis, and for the purpose of improving the state of the art in software verification and vulnerability analysis.

8.1 Contributions

In Chapter 2 we brought to light the false positives problem which appears when applying Symbolic Execution to vulnerability analysis. We first showed how on an authentication bypass scenario a C-level classic Symbolic Execution answers with an incorrect solution, due to the absence of distinction between controlled and uncontrolled inputs. Then we demonstrated how this issue can be overcome by switching to a binary-level semantics and a better modeling of interactions with the environment taking care of controlled and uncontrolled inputs, but at the price of generating larger formulas involving quantifiers.

After presenting the many-sorted first-order logic in Chapter 3 and Symbolic Execution in Chapter 4, we addressed in Chapter 5 the problem of generating models of quantified first-order formulas over built-in theories. We proposed a novel and generic taint-based approach relying on a reduction to the quantifier-free case through

the inference of independence conditions, and proved its correctness and its efficiency under reasonable assumptions. This technique is applicable to any theory with a decidable quantifier-free case and allows to reuse all the work done on quantifier-free solvers. We presented a concrete implementation of our method specialized on arrays and bit-vectors that we evaluate on SMT-LIB benchmarks and formulas generated by the binary-level Symbolic Execution tool Binsec. The method significantly enhances the performances of state-of-the-art SMT solvers for the quantified case, and supplements the latest advances in the field.

Then we presented in Chapter 6 fas, an original simplification method dedicated to the theory of arrays, a theory which has a central place in software verification due to its ability to model memory or data structures. As this theory is known to be hard to solve because of *read-over-write* terms, fasis geared at eliminating row, based on a new dedicated data structure together with original simplifications and low-cost reasoning. We experimentally evaluated it in different settings and we showed that the technique is efficient, scalable and yields significant simplifications. The impact on formula resolution is always positive, and it can be dramatic on formulas generated by binary-level Symbolic Execution.

Finally in Chapter 7 we formally introduced Robust Symbolic Execution, a new framework which makes possible to remedy definitively the false positives problem, provided that solvers manage to handle large generated quantified formulas. This is achieved thanks to our recent work on taint-based quantifier elimination and array terms simplification presented in the previous chapters. A proof of concept is implemented in the binary analyzer Binsec, resulting in the first Robust Symbolic Execution tool really exempt of false positives.

8.2 Perspectives

Some notable research prospects extend directly the work developed in this thesis, amongst them the improvement of the following aspects:

Concerning model generation for quantified formulas, future work aims to tackle the definition of more precise inference mechanisms of independence conditions, the identification of interesting subclasses for which inferring weakest independence conditions is feasible, and the combination with other quantifier instantiation techniques;

Concerning simplification of array terms, future work includes a deeper integration inside a dedicated array solver in order to benefit from more simplification

opportunities along the resolution process, as well as exploring the interest of adding more expressive domain reasoning;

Finally, concerning Robust Symbolic Execution, future work comprises a precise evaluation of our semi-automatic incremental specification procedure for inputs, and a thorough comparison of Robust Symbolic Execution to other techniques.

Beyond these extensions, even more interesting research prospects can be found by widening our problematic. We have seen in this thesis how formalizing some security properties using first-order logic requires the use of quantifiers, for example when we want to find an input which allows to bypass a guard condition whatever the value of a non-deterministic uncontrollable variable is. However, to restrict oneself to the usual "there exists" and "for all" quantifications is not nuanced enough if we want to formalize a property which says that we can bypass a guard condition "almost always", or that an event "almost never" occurs. Such nuances can be formalized by introducing notions of probabilities, or by using model counting.

The model counting problem consists in finding the number of models which satisfy a given formula. But solving the model counting problem is even harder than solving the satisfiability problem. Fortunately for many properties, finding an upper or lower bound which approximates the solution to the model counting problem is sufficient to conclude: we can find an input which bypasses a guard condition in more than N_{inf} cases, or we can prove that an event only occurs in less than N_{sup} cases. Finding non-trivial lower and upper bounds of the number of models for a given formula remains a difficult problem, but is nevertheless more achievable than finding the exact number of models, and therefore gives an interesting line of research which would pave the way for new advances in formal verification.

Bibliography

[And72] James P. Anderson. Computer security technology planning study. Technical Report ESD-TR-73-51, Vol.II, Air Force Electronic Systems Division, 1972.

[ARCB14] Thanassis Avgerinos, Alexandre Rebert, Sang Kil Cha, and David Brumley. Enhancing symbolic execution with veritesting. In *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*, pages 1083–1094, 2014.

[Bar16] Haniel Barbosa. Efficient Instantiation Techniques in SMT (work in progress). In *Proceedings of the 5th Workshop on Practical Aspects of Automated Reasoning co-located with International Joint Conference on Automated Reasoning (IJCAR 2016), Coimbra, Portugal, July 2nd, 2016.*, pages 1–10, 2016.

[BB09a] Robert Brummayer and Armin Biere. Boolector: An Efficient SMT Solver for Bit-Vectors and Arrays. In *Tools and Algorithms for the Construction and Analysis of Systems, 15th International Conference, TACAS 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings*, pages 174–177, 2009.

[BB09b] Robert Brummayer and Armin Biere. Lemmas on demand for the extensional theory of arrays. *JSAT*, 6(1-3):165–201, 2009.

[BBD+17] Benjamin Beurdouche, Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, Pierre-Yves Strub, and Jean Karim Zinzindohoue. A messy state of the union: taming the composite state machines of TLS. *Commun. ACM*, 60(2):99–107, 2017.

[BBK17] Karthikeyan Bhargavan, Bruno Blanchet, and Nadim Kobeissi. Verified models and reference implementations for the TLS 1.3 standard candidate.

In *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*, pages 483–502, 2017.

[BBKK12] Sebastian Biallas, Jörg Brauer, Andy King, and Stefan Kowalewski. Loop leaping with closures. In *Static Analysis - 19th International Symposium, SAS 2012, Deauville, France, September 11-13, 2012. Proceedings*, pages 214–230, 2012.

[BCD+11] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, pages 171–177, 2011.

[BCE08] Peter Boonstoppel, Cristian Cadar, and Dawson R. Engler. Rwset: Attacking path explosion in constraint-based test generation. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, pages 351–366, 2008.

[BCF+07] Roberto Bruttomesso, Alessandro Cimatti, Anders Franzén, Alberto Griggio, Ziyad Hanna, Alexander Nadel, Amit Palti, and Roberto Sebastiani. A lazy and layered SMT(BV) solver for hard industrial verification problems. In *Computer Aided Verification, 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007, Proceedings*, pages 547–560, 2007.

[BDL98] Clark W. Barrett, David L. Dill, and Jeremy R. Levitt. A decision procedure for bit-vector arithmetic. In *Proceedings of the 35th Conference on Design Automation, Moscone center, San Francico, California, USA, June 15-19, 1998.*, pages 522–527, 1998.

[BDM17] Sébastien Bardin, Robin David, and Jean-Yves Marion. Backward-bounded DSE: targeting infeasibility questions on obfuscated codes. In *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*, pages 633–651. IEEE, 2017.

[BFK16] Karthikeyan Bhargavan, Cédric Fournet, and Markulf Kohlweiss. mitls: Verifying protocol implementations against real-world attacks. *IEEE Security & Privacy*, 14(6):18–25, 2016.

[BG94] Leo Bachmair and Harald Ganzinger. Rewrite-Based Equational Theorem Proving with Selection and Simplification. *J. Log. Comput.*, 4(3):217–247, 1994.

[BG12] Sébastien Bardin and Arnaud Gotlieb. FDCC: A combined approach for solving constraints over finite domains and arrays. In *Integration of AI and OR Techniques in Contraint Programming for Combinatorial Optimzation Problems - 9th International Conference, CPAIOR 2012, Nantes, France, May 28 - June1, 2012. Proceedings*, pages 17–33, 2012.

[BGM13] Ella Bounimova, Patrice Godefroid, and David A. Molnar. Billions and billions of constraints: whitebox fuzz testing in production. In *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, pages 122–131, 2013.

[BHP10] Sébastien Bardin, Philippe Herrmann, and Florian Perroud. An alternative to sat-based approaches for bit-vectors. In *Tools and Algorithms for the Construction and Analysis of Systems, 16th International Conference, TACAS 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings*, pages 84–98, 2010.

[Bie09] Armin Biere. Bounded Model Checking. In *Handbook of Satisfiability*, pages 457–481. 2009.

[BJAS11] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J. Schwartz. BAP: A binary analysis platform. In *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, pages 463–469, 2011.

[Bjø10] Nikolaj Bjørner. Linear quantifier elimination as an abstract decision procedure. In *Automated Reasoning, 5th International Joint Conference, IJCAR 2010, Edinburgh, UK, July 16-19, 2010. Proceedings*, pages 316–330, 2010.

[BKO+07] Randal E. Bryant, Daniel Kroening, Joël Ouaknine, Sanjit A. Seshia, Ofer Strichman, and Bryan A. Brady. Deciding bit-vector arithmetic with abstraction. In *Tools and Algorithms for the Construction and Analysis of Systems, 13th International Conference, TACAS 2007, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2007 Braga, Portugal, March 24 - April 1, 2007, Proceedings*, pages 358–372, 2007.

[BKRW11] Angelo Brillout, Daniel Kroening, Philipp Rümmer, and Thomas Wahl. Beyond Quantifier-Free Interpolation in Extensions of Presburger Arithmetic. In *Verification, Model Checking, and Abstract Interpretation - 12th International Conference, VMCAI 2011, Austin, TX, USA, January 23-25, 2011. Proceedings*, pages 88–102, 2011.

[BM07] Aaron R. Bradley and Zohar Manna. *The Calculus of Computation: Decision Procedures with Applications to Verification (Chapters 5, 6 and 12)*. Springer, 2007.

[BM18] Sylvie Boldo and Nicolas Magaud, editors. *Journées Francophones des Langages Applicatifs, JFLA 2018, Banyuls-sur-Mer, France, January 24-27, 2018*, 2018.

[BMS06] Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. What's decidable about arrays? In *Verification, Model Checking, and Abstract Interpretation, 7th International Conference, VMCAI 2006, Charleston, SC, USA, January 8-10, 2006, Proceedings*, pages 427–442, 2006.

[BNO+08] Miquel Bofill, Robert Nieuwenhuis, Albert Oliveras, Enric Rodríguez-Carbonell, and Albert Rubio. A write-based solver for SAT modulo the theory of arrays. In *Formal Methods in Computer-Aided Design, FMCAD 2008, Portland, Oregon, USA, 17-20 November 2008*, pages 1–8, 2008.

[BRK+15] Kshitij Bansal, Andrew Reynolds, Tim King, Clark W. Barrett, and Thomas Wies. Deciding local theory extensions via e-matching. In *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part II*, pages 87–105, 2015.

[BSST09] Clark W. Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. Satisfiability Modulo Theories. In *Handbook of Satisfiability*, pages 825–885. 2009.

[BST10] Clark Barrett, Aaron Stump, and Cesare Tinelli. The SMT-LIB Standard: Version 2.0. In A. Gupta and D. Kroening, editors, *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, UK)*, 2010.

[BT18] Clark W. Barrett and Cesare Tinelli. Satisfiability modulo theories. In *Handbook of Model Checking.*, pages 305–343. 2018.

[Cad15] Cristian Cadar. Targeted program transformations for symbolic execution. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*, pages 906–909, 2015.

[CARB12] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. Unleashing mayhem on binary code. In *IEEE Symposium on Security and Privacy, SP 2012, 21-23 May 2012, San Francisco, California, USA*, pages 380–394, 2012.

[CC77] Patrick Cousot and Radia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *4th ACM Symposium on Principles of Programming Languages (POPL)*. ACM, 1977.

[CDE08] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*, pages 209–224, 2008.

[CF00] Thomas Colcombet and Pascal Fradet. Enforcing trace properties by program transformation. In *POPL 2000, Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Boston, Massachusetts, USA, January 19-21, 2000*, pages 54–66, 2000.

[cgc16] Cyber grand challenge (cgc). https://www.darpa.mil/program/cyber-grand-challenge, 2016.

[CGP+06] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. EXE: automatically generating inputs of death. In *Proceedings of the 13th ACM Conference on Computer and Communications Security, CCS 2006, Alexandria, VA, USA, Ioctober 30 - November 3, 2006*, pages 322–335, 2006.

[CH88] Thierry Coquand and Gérard P. Huet. The calculus of constructions. *Inf. Comput.*, 76(2/3):95–120, 1988.

[CKC11] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. S2E: a platform for in-vivo multi-path analysis of software systems. In *Proceedings*

of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2011, Newport Beach, CA, USA, March 5-11, 2011, pages 265–278, 2011.

[CKC12] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. The S2E platform: Design, implementation, and applications. *ACM Trans. Comput. Syst.*, 30(1):2:1–2:49, 2012.

[CKGJ11] Omar Chebaro, Nikolai Kosmatov, Alain Giorgetti, and Jacques Julliand. The SANTE tool: Value analysis, program slicing and test generation for C program debugging. In *Tests and Proofs - 5th International Conference, TAP 2011, Zurich, Switzerland, June 30 - July 1, 2011. Proceedings*, pages 78–83, 2011.

[CKL04] Edmund M. Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In *TACAS, Barcelona, Spain, March 29 - April 2, 2004*, pages 168–176, 2004.

[Cla76] Lori A. Clarke. A system to generate test data and symbolically execute programs. *IEEE Trans. Software Eng.*, 2(3):215–222, 1976.

[CMMN12] Christian S. Collberg, Sam Martin, Jonathan Myers, and Jasvir Nagra. Distributed application tamper detection via continuous software updates. In *28th Annual Computer Security Applications Conference, ACSAC 2012, Orlando, FL, USA, 3-7 December 2012*, pages 319–328, 2012.

[CS13] Cristian Cadar and Koushik Sen. Symbolic execution for software testing: three decades later. *Commun. ACM*, 56(2):82–90, 2013.

[CW96] Edmund M. Clarke and Jeannette M. Wing. Formal methods: State of the art and future directions. *ACM Comput. Surv.*, 28(4):626–643, 1996.

[DB15] Adel Djoudi and Sébastien Bardin. BINSEC: binary code analysis with low-level regions. In *Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, pages 212–217, 2015.

[DBF+16] Robin David, Sébastien Bardin, Josselin Feist, Laurent Mounier, Marie-Laure Potet, Thanh Dinh Ta, and Jean-Yves Marion. Specification of

concretization and symbolization policies in symbolic execution. In *ISSTA, Saarbrücken, Germany, July 18-20, 2016*, pages 36–46, 2016.

[DBG10] Mickaël Delahaye, Bernard Botella, and Arnaud Gotlieb. Explanation-based generalization of infeasible path. In *Third International Conference on Software Testing, Verification and Validation, ICST 2010, Paris, France, April 7-9, 2010*, pages 215–224, 2010.

[DBT+16] Robin David, Sébastien Bardin, Thanh Dinh Ta, Laurent Mounier, Josselin Feist, Marie-Laure Potet, and Jean-Yves Marion. BINSEC/SE: A Dynamic Symbolic Execution Toolkit for Binary-Level Analysis. In *IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering, SANER 2016, Osaka, Japan, March 14-18, 2016 - Volume 1*, pages 653–656, 2016.

[DCKP12] Claire Dross, Sylvain Conchon, Johannes Kanig, and Andrei Paskevich. Reasoning with Triggers. In *10th International Workshop on Satisfiability Modulo Theories, SMT 2012, Manchester, UK, June 30 - July 1, 2012*, pages 22–31, 2012.

[DCKP16] Claire Dross, Sylvain Conchon, Johannes Kanig, and Andrei Paskevich. Adding Decision Procedures to SMT Solvers Using Axioms with Triggers. *J. Autom. Reasoning*, 56(4):387–457, 2016.

[DD77] Dorothy E. Denning and Peter J. Denning. Certification of Programs for Secure Information Flow. *Commun. ACM*, 20(7):504–513, 1977.

[DKA+14] Zakir Durumeric, James Kasten, David Adrian, J. Alex Halderman, Michael Bailey, Frank Li, Nicholas Weaver, Johanna Amann, Jethro Beekman, Mathias Payer, and Vern Paxson. The matter of heartbleed. In *2014 Internet Measurement Conference, IMC 2014, Vancouver, Canada, November 5-7, 2014*, pages 475–488, 2014.

[DLL62] Martin Davis, George Logemann, and Donald W. Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, 1962.

[dMB07] Leonardo Mendonça de Moura and Nikolaj Bjørner. Efficient e-matching for SMT solvers. In *Automated Deduction - CADE-21, 21st International Conference on Automated Deduction, Bremen, Germany, July 17-20, 2007, Proceedings*, pages 183–198, 2007.

[dMB08] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, pages 337–340, 2008.

[DMR08] Ádám Darvas, Farhad Mehta, and Arsenii Rudich. Efficient Well-Definedness Checking. In *Automated Reasoning, 4th International Joint Conference, IJCAR 2008, Sydney, Australia, August 12-15, 2008, Proceedings*, pages 100–115, 2008.

[DNS05] David Detlefs, Greg Nelson, and James B. Saxe. Simplify: a theorem prover for program checking. *J. ACM*, 52(3):365–473, 2005.

[dOBP16] Steven de Oliveira, Saddek Bensalem, and Virgile Prevosto. Polynomial invariants by linear algebra. In *Automated Technology for Verification and Analysis - 14th International Symposium, ATVA 2016, Chiba, Japan, October 17-20, 2016, Proceedings*, pages 479–494, 2016.

[DP60] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *J. ACM*, 7(3):201–215, 1960.

[DS78] Peter J. Downey and Ravi Sethi. Assignment commands with array references. *J.ACM*, 25(4):652–666, 1978.

[Dut14] Bruno Dutertre. Yices 2.2. In *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, pages 737–744, 2014.

[FBBP18] Benjamin Farinier, Sébastien Bardin, Richard Bonichon, and Marie-Laure Potet. Model generation for quantified formulas: A taint-based approach. In *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part II*, pages 294–313, 2018.

[FC06] Jean-Christophe Filliâtre and Sylvain Conchon. Type-safe modular hash-consing. In *Proceedings of the ACM Workshop on ML, 2006, Portland, Oregon, USA, September 16, 2006*, pages 12–19, 2006.

[FDBL18] Benjamin Farinier, Robin David, Sébastien Bardin, and Matthieu Lemerre. Arrays made simpler: An efficient, scalable and thorough preprocessing.

In *LPAR-22. 22nd International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Awassa, Ethiopia, 16-21 November 2018*, pages 363–380, 2018.

[FK16] Azadeh Farzan and Zachary Kincaid. Linear Arithmetic Satisfiability via Strategy Improvement. In *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence, IJCAI 2016, New York, NY, USA, 9-15 July 2016*, pages 735–743, 2016.

[FLP16] Aymeric Fromherz, Kasper Søe Luckow, and Corina S. Pasareanu. Symbolic arrays in symbolic pathfinder. *ACM SIGSOFT Software Engineering Notes*, 41(6):1–5, 2016.

[GD07] Vijay Ganesh and David L. Dill. A decision procedure for bit-vectors and arrays. In *Computer Aided Verification, 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007, Proceedings*, pages 519–531, 2007.

[GdM09] Yeting Ge and Leonardo Mendonça de Moura. Complete instantiation for quantified formulas in satisfiabiliby modulo theories. In *Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings*, pages 306–320, 2009.

[GHN+04] Harald Ganzinger, George Hagen, Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. DPLL(T): fast decision procedures. In *Computer Aided Verification, 16th International Conference, CAV 2004, Boston, MA, USA, July 13-17, 2004, Proceedings*, pages 175–188, 2004.

[GKS05] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: directed automated random testing. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12-15, 2005*, pages 213–223, 2005.

[GL11] Patrice Godefroid and Daniel Luchaup. Automatic partial loop summarization in dynamic test generation. In *Proceedings of the 20th International Symposium on Software Testing and Analysis, ISSTA 2011, Toronto, ON, Canada, July 17-21, 2011*, pages 23–33, 2011.

[GLM12] Patrice Godefroid, Michael Y. Levin, and David A. Molnar. SAGE: Whitebox Fuzzing for Security Testing. *ACM Queue*, 10(1):20, 2012.

[God07] Patrice Godefroid. Compositional dynamic test generation. In *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2007, Nice, France, January 17-19, 2007*, pages 47–54, 2007.

[hea14] The heartbleed bug. http://heartbleed.com/, 2014.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[IJS08] Carsten Ihlemann, Swen Jacobs, and Viorica Sofronie-Stokkermans. On local reasoning in verification. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, pages 265–281, 2008.

[JS97] Roberto J. Bayardo Jr. and Robert Schrag. Using CSP look-back techniques to solve real-world SAT instances. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence and Ninth Innovative Applications of Artificial Intelligence Conference, AAAI 97, IAAI 97, July 27-31, 1997, Providence, Rhode Island, USA.*, pages 203–208, 1997.

[JS16] Martin Jonás and Jan Strejcek. Solving Quantified Bit-Vector Formulas Using Binary Decision Diagrams. In *Theory and Applications of Satisfiability Testing - SAT 2016 - 19th International Conference, Bordeaux, France, July 5-8, 2016, Proceedings*, pages 267–283, 2016.

[Kin76] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.

[KKBC12] Volodymyr Kuznetsov, Johannes Kinder, Stefan Bucur, and George Candea. Efficient state merging in symbolic execution. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*, pages 193–204, 2012.

[KKP+15] Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-c: A software analysis perspective. *Formal Asp. Comput.*, 27(3):573–609, 2015.

[KS08] Daniel Kroening and Ofer Strichman. *Decision Procedures - An Algorithmic Point of View*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2008.

[KV13] Laura Kovács and Andrei Voronkov. First-Order Theorem Proving and Vampire. In *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, pages 1–35, 2013.

[MD19] Nicolas Magaud and Zaynah Dargaye, editors. *Journées Francophones des Langages Applicatifs, JFLA 2019, Les Rousses, France, January 30-February 2, 2019*, 2019.

[MR15] Hector Marco and Ismael Ripoll. Back to 28: Grub2 authentication 0-day. http://hmarco.org/bugs/CVE-2015-8370-Grub2-authentication-bypass.html, 2015.

[MV07] Panagiotis Manolios and Daron Vroon. Efficient circuit to CNF conversion. In *Theory and Applications of Satisfiability Testing - SAT 2007, 10th International Conference, Lisbon, Portugal, May 28-31, 2007, Proceedings*, pages 4–9, 2007.

[NO79] Greg Nelson and Derek C. Oppen. Simplification by cooperating decision procedures. *ACM Trans. Program. Lang. Syst.*, 1(2):245–257, 1979.

[NO05] Robert Nieuwenhuis and Albert Oliveras. Proof-producing congruence closure. In *Term Rewriting and Applications, 16th International Conference, RTA 2005, Nara, Japan, April 19-21, 2005, Proceedings*, pages 453–468, 2005.

[NR01] Robert Nieuwenhuis and Albert Rubio. Paramodulation-Based Theorem Proving. In *Handbook of Automated Reasoning (in 2 volumes)*, pages 371–443. 2001.

[NS05] James Newsome and Dawn Xiaodong Song. Dynamic taint analysis for automatic detection, analysis, and signaturegeneration of exploits on commodity software. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2005, San Diego, California, USA*, 2005.

[Opp80] Derek C. Oppen. Complexity, convexity and combinations of theories. *Theor. Comput. Sci.*, 12:291–302, 1980.

[Ørb95] Peter Ørbæk. Can you Trust your Data? In *TAPSOFT'95: Theory and Practice of Software Development, 6th International Joint Conference CAAP/FASE, Aarhus, Denmark, May 22-26, 1995, Proceedings*, pages 575–589, 1995.

[PMZC17] David Mitchel Perry, Andrea Mattavelli, Xiangyu Zhang, and Cristian Cadar. Accelerating array constraints in symbolic execution. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, Santa Barbara, CA, USA, July 10 - 14, 2017*, pages 68–78, 2017.

[PNB17] Mathias Preiner, Aina Niemetz, and Armin Biere. Counterexample-Guided Model Synthesis. In *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part I*, pages 264–280, 2017.

[RDK+15] Andrew Reynolds, Morgan Deters, Viktor Kuncak, Cesare Tinelli, and Clark W. Barrett. Counterexample-guided quantifier instantiation for synthesis in SMT. In *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part II*, pages 198–216, 2015.

[RL18] Antoine Rollet and Arnaud Lanoix, editors. *Approches Formelles dans l'Assistance au Développement de Logiciels, AFADL 2018, Grenoble, France, June 13-15, 2018*, 2018.

[RTdM14] Andrew Reynolds, Cesare Tinelli, and Leonardo Mendonça de Moura. Finding conflicting instances of quantified formulas in SMT. In *Formal Methods in Computer-Aided Design, FMCAD 2014, Lausanne, Switzerland, October 21-24, 2014*, pages 195–202, 2014.

[RTG+13] Andrew Reynolds, Cesare Tinelli, Amit Goel, Sava Krstic, Morgan Deters, and Clark Barrett. Quantifier instantiation techniques for finite model finding in SMT. In *Automated Deduction - CADE-24 - 24th International Conference on Automated Deduction, Lake Placid, NY, USA, June 9-14, 2013. Proceedings*, pages 377–391, 2013.

[RTGK13] Andrew Reynolds, Cesare Tinelli, Amit Goel, and Sava Krstic. Finite model finding in SMT. In *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, pages 640–655, 2013.

[Rus05] John M. Rushby. Automated test generation and verified software. In *Verified Software: Theories, Tools, Experiments, First IFIP TC 2/WG 2.3 Conference, VSTTE 2005, Zurich, Switzerland, October 10-13, 2005, Revised Selected Papers and Discussions*, pages 161–172, 2005.

[RV02] Alexandre Riazanov and Andrei Voronkov. The design and implementation of VAMPIRE. *AI Commun.*, 15(2-3):91–110, 2002.

[SAB10] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *31st IEEE Symposium on Security and Privacy, S&P 2010, 16-19 May 2010, Berleley/Oakland, California, USA*, pages 317–331, 2010.

[SBP18] Jonathan Salwan, Sébastien Bardin, and Marie-Laure Potet. Symbolic deobfuscation: From virtualized code back to the original. In *Detection of Intrusions and Malware, and Vulnerability Assessment - 15th International Conference, DIMVA 2018, Saclay, France, June 28-29, 2018, Proceedings*, pages 372–392. Springer, 2018.

[SBY+08] Dawn Xiaodong Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Prateek Saxena. Bitblaze: A new approach to computer security via binary analysis. In *Information Systems Security, 4th International Conference, ICISS 2008, Hyderabad, India, December 16-20, 2008. Proceedings*, pages 1–25, 2008.

[Sch02] Stephan Schulz. E - a brainiac theorem prover. *AI Commun.*, 15(2-3):111–126, 2002.

[Sho78] Robert E. Shostak. An algorithm for reasoning about equality. *Commun. ACM*, 21(7):583–585, 1978.

[Sim08] Axel Simon. *Value-Range Analysis of C Programs: Towards Proving the Absence of Buffer Overflow Vulnerabilities*. Springer, 2008.

[SMA05] Koushik Sen, Darko Marinov, and Gul Agha. CUTE: a concolic unit testing engine for C. In *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2005, Lisbon, Portugal, September 5-9, 2005*, pages 263–272, 2005.

[Smi07] Geoffrey Smith. Principles of Secure Information Flow Analysis. In *Malware Detection*, pages 291–307. 2007.

[SS96] João P. Marques Silva and Karem A. Sakallah. GRASP - a new search algorithm for satisfiability. In *ICCAD*, pages 220–227, 1996.

[SS99] João P. Marques Silva and Karem A. Sakallah. GRASP: A search algorithm for propositional satisfiability. *IEEE Trans. Computers*, 48(5):506–521, 1999.

[SSJ+15] Daniel Schwartz-Narbonne, Martin Schäf, Dejan Jovanovic, Philipp Rmmer, and Thomas Wies. Conflict-directed graph coverage. In *NASA Formal Methods - 7th International Symposium, NFM 2015, Pasadena, CA, USA, April 27-29, 2015, Proceedings*, pages 327–342, 2015.

[SST13] Jiri Slaby, Jan Strejcek, and Marek Trtík. Compact symbolic execution. In *Automated Technology for Verification and Analysis - 11th International Symposium, ATVA 2013, Hanoi, Vietnam, October 15-18, 2013. Proceedings*, pages 193–207, 2013.

[TH96] Cesare Tinelli and Mehdi T. Harandi. A new correctness proof of the nelson-oppen combination procedure. In *Frontiers of Combining Systems, First International Workshop FroCoS 1996, Munich, Germany, March 26-29, 1996, Proceedings*, pages 103–119, 1996.

[Tin02] Cesare Tinelli. A DPLL-based calculus for ground satisfiability modulo theories. In *Logics in Artificial Intelligence, European Conference, JELIA 2002, Cosenza, Italy, September, 23-26, Proceedings*, pages 308–319, 2002.

[Tip95] Frank Tip. A survey of program slicing techniques. *J. Prog. Lang.*, 3(3), 1995.

[Wei81] Mark Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering, San Diego, California, USA, March 9-12, 1981.*, pages 439–449, 1981.

[WHdM10] Christoph M. Wintersteiger, Youssef Hamadi, and Leonardo Mendonça de Moura. Efficiently solving quantified bit-vector formulas. In *Proceedings of 10th International Conference on Formal Methods in Computer-Aided Design, FMCAD 2010, Lugano, Switzerland, October 20-23*, pages 239–246, 2010.

[WMMR05] Nicky Williams, Bruno Marre, Patricia Mouy, and Muriel Roger. Pathcrawler: Automatic generation of path tests by combining static and dynamic analysis. In *Dependable Computing - EDCC-5, 5th European Dependable Computing Conference, Budapest, Hungary, April 20-22, 2005, Proceedings*, pages 281–292, 2005.

[WSK05] Markus Wedler, Dominik Stoffel, and Wolfgang Kunz. Normalization at the arithmetic bit level. In *Proceedings of the 42nd Design Automation Conference, DAC 2005, San Diego, CA, USA, June 13-17, 2005*, pages 457–462, 2005.

[YJWD15] Babak Yadegari, Brian Johannesmeyer, Ben Whitely, and Saumya Debray. A generic approach to automatic deobfuscation of executable code. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*, pages 674–691. IEEE, 2015.

Résumé

L'Exécution symbolique est une technique de vérification formelle qui consiste en modéliser les exécutions d'un programme par des formules logiques pour montrer que ces exécutions vérifient une propriété donnée. Très efficace pour la recherche de bogues, il est question aujourd'hui de l'employer dans d'autres contextes, comme en analyse de vulnérabilités. L'application de l'Exécution symbolique à l'analyse de vulnérabilités diffère de la recherche de bogues sur au moins deux aspects :

- Les formules logiques générées au cours de l'Exécution symbolique deviennent rapidement gigantesques et de plus en plus difficiles à résoudre pour les solveurs.
- La modélisation de certaines propriétés de sécurité est susceptible de faire intervenir des quantificateurs dont l'emploi rend les formules logiques générées presque impossibles à résoudre.

Cette thèse porte donc sur ces deux problématiques issues du domaine des procédures de décision, visant à permettre des modélisations plus fines nécessaires à l'analyse de vulnérabilités.

Abstract

Symbolic Execution is a formal verification technique which consists in modeling program executions by logical formulas in order to prove that these executions verify a given property. Very effective for bug finding, the question is now how to use it in other contexts, for example in vulnerability analysis. Applying Symbolic Execution to vulnerability analysis fundamentally differs from bug finding on at least two aspects:

- Logical formulas generated during an analysis quickly become gigantic, and more and more difficult to solve.
- Modeling some security properties is likely to involve quantifiers whose use made generated logical formulas nearly impossible to solve.

Therefore this thesis focuses on these two issues arising from the field of decision procedures, in order to allow finer models required for vulnerability analysis.