Robust reachability and model counting for software security

Guillaume Girol

université
PARIS-SACLAY

Robust reachability and model counting for software security

Atteignabilité robuste et comptage de modèles pour la sécurité logicielle

Thèse de doctorat de l'université Paris-Saclay

École doctorale n° 580, Sciences et Technologies de l'Information et de la Communication (STIC)
Spécialité de doctorat : Informatique
Graduate School : Informatique et sciences du numérique, Référent : Faculté des sciences d'Orsay

Thèse préparée dans les unités de recherche **Laboratoire Méthodes Formelles** (Université Paris-Saclay, CNRS, ENS Paris-Saclay) et **Institut List** (Université Paris-Saclay, CEA), sous la direction de **Sylvain CONCHON**, professeur à l'Université Paris-Saclay, et le co-encadrement de **Sébastien BARDIN**, chercheur à l'Institut CEA LIST.

Thèse soutenue à Paris-Saclay, le 17 octobre 2022, par

Guillaume GIROL

Composition du jury

Mihaela SIGHIREANU Professeure, Université Paris-Saclay	Présidente
Ahmed BOUAJJANI Professeur, Université Paris Cité	Rapporteur & Examinateur
Roberto GIACOBAZZI Professeur, Université de Vérone	Rapporteur & Examinateur
Thomas JENSEN Directeur de Recherche, INRIA – Université de Rennes 1	Examinateur
Caterina URBAN Chargée de Recherche, INRIA – Université Paris Sciences Lettres	Examinatrice
Sylvain CONCHON Professeur, Université Paris-Saclay	Directeur de thèse

THESE DE DOCTORAT
NNT : 2022UPASG071

université PARIS-SACLAY

ÉCOLE DOCTORALE
Sciences et technologies de l'information et de la communication (STIC)

Titre : Atteignabilité robuste et comptage de modèles pour la sécurité logicielle
Mots clés : robustesse, évaluation de vulnerabilité, sécurité, comptage de modèles, exécution symbolique

Résumé : Les techniques modernes de recherche de bugs sont devenues si efficaces que le problème n'est plus de trouver des bugs mais de trouver le temps de les corriger. Une façon répandue d'éluder ce problème est de concentrer l'effort de correction de bug prioritairement sur les bugs ayant un impact en termes de sécurité, aussi désignés sous le nom de vulnérabilités. Cela conduit naturellement à la question de l'évaluation de cet impact : un attaquant pourrait-il tirer parti de tel ou tel bug ? Cette thèse se concentre sur une dimension particulière de ce problème : un attaquant serait-il capable de déclencher ce bug ? Nous appelons cette propriété la réplicabilité. Nous nous fixons pour objectif de concevoir des méthodes de recherches de bugs qui ne détectent que des bugs suffisamment réplicables. Le point de départ est de considérer des modèles de menace où l'on distingue les entrées du programme qui peuvent être choisies par l'attaquant (comme les entrées réseau) de celle qui ne peuvent ni être contrôlées ni connues de lui (comme les sources d'entropie). Nous proposons deux approches pour évaluer la réplicabilité. D'abord, nous définissons l'atteignabilité robuste, une propriété qualitative qui exprime qu'un bug est non seulement atteignable mais que lorsqu'il choisit les entrées qu'il peut correctement, l'attaquant déclenche toujours le bug, indépendamment des valeurs des entrées qu'il ne peut pas choisir. Dans un second temps, nous affinons cette approche en une approche quantitative où nous déterminons la proportion d'entrées non contrôlées qui permettent à l'attaquant optimal de déclencher le bug. Nous adaptons ensuite l'exécution symbolique pour prouver l'atteignabilité robuste ou bien calculer cette proportion. L'atteignabilité robuste est une analyse moins fine que cette approche quantitative parce qu'elle est "tout ou rien", mais en contrepartie elle passe mieux à l'échelle. Enfin nous illustrons dans des études de cas les applications potentielles de ces concepts, en particulier à l'évaluation de vulnérabilité.

Title: Robust reachability and model counting for software security
Keywords: robustness, vulnerability assessment, security, model counting, symbolic execution

Abstract: Modern bug-finding techniques have become effective enough that the bottleneck is not finding bugs but finding the time to fix them. A popular way to address this problem is to focus first on bugs with a security impact, also known as vulnerabilities. This leads to the question of vulnerability assessment: could an attacker take advantage of a bug? In this thesis we attempt to assess one particular dimension contributing to the security impact of a bug: whether an attacker could trigger it reliably. We call this property *replicability*. Our goal is to formalize replicability to design bug-finding techniques which only report bugs which are replicable enough. We do so by considering a threat model where inputs to the program which the attacker can choose (like network inputs) are distinguished from inputs which the attacker does not control nor know (like entropy sources). We propose two approaches to replicability. Firstly, we define robust reachability, a qualitative property that expresses that a bug is not only reachable, but that when he chooses the right inputs, the attacker triggers the bug whatever the values of the program inputs he does not control. Secondly, we refine robust reachability quantitatively as the proportion of uncontrolled inputs that let the optimal attacker trigger the bug. We adapt symbolic execution to prove robust reachability and compute this proportion. Robust reachability is more coarse-grained because it is "all-or-nothing" but scales better than the quantitative approach. We illustrate in case studies the potential applications of these techniques, notably in terms of vulnerability assessment.

Order Number: 2022UPASG071

Université Paris-Saclay

Doctoral School **STIC**

University Department **CEA List/LSL**

Thesis defended by **Guillaume Girol**

Defended on **October 17, 2022**

In order to become Doctor from Université Paris-Saclay

Academic Field **Computer Science**

Robust reachability and model counting for software security

Thesis supervised by

Sylvain Conchon	Supervisor
Sébastien Bardin	Co-Monitor

Committee members

Referees	Ahmed Bouajjani	Professor at Université Paris Cité	
	Roberto Giacobazzi	Professor at University of Verona	
Examiners	Thomas Jensen	Senior Researcher at INRIA — Université de Rennes 1	
	Mihaela Sighireanu	Professor at Université Paris-Saclay	Committee President
	Caterina Urban	Research Scientist at INRIA — Université Paris Sciences Lettres	
Supervisor	Sylvain Conchon	Professor at Laboratoire Méthodes Formelles, Université Paris-Saclay	

Université Paris-Saclay

École doctorale **STIC**

Unité de recherche **CEA List/LSL**

Thèse présentée par **Guillaume Girol**

Soutenue le **17 octobre 2022**

En vue de l'obtention du grade de docteur de l'Université Paris-Saclay

Discipline **Informatique**

Atteignabilité robuste et comptage de modèles pour la sécurité logicielle

Thèse dirigée par

Sylvain Conchon	directeur
Sébastien Bardin	co-encadrant

Composition du jury

Rapporteurs	Ahmed Bouajjani	Professeur à l'Université Paris Cité	
	Roberto Giacobazzi	Professeur à l'Université de Vérone	
Examinateurs	Thomas Jensen	Directeur de Recherche à l'INRIA — Université de Rennes 1	
	Mihaela Sighireanu	Professeure à l'Université Paris-Saclay	présidente du jury
	Caterina Urban	Chargée de Recherche à l'INRIA — Université Paris Sciences Lettres	
Directeur de thèse	Sylvain Conchon	Professeur au Laboratoire Méthodes Formelles, Université Paris-Saclay	

Robust reachability and model counting for software security

Abstract

Modern bug-finding techniques have become effective enough that the bottleneck is not finding bugs but finding the time to fix them. A popular way to address this problem is to focus first on bugs with a security impact, also known as vulnerabilities. This leads to the question of vulnerability assessment: could an attacker take advantage of a bug? In this thesis we attempt to assess one particular dimension contributing to the security impact of a bug: whether an attacker could trigger it reliably. We call this property *replicability*. Our goal is to formalize replicability to design bug-finding techniques which only report bugs which are replicable enough. We do so by considering a threat model where inputs to the program which the attacker can choose (like network inputs) are distinguished from inputs which the attacker does not control nor know (like entropy sources). We propose two approaches to replicability. Firstly, we define robust reachability, a qualitative property that expresses that a bug is not only reachable, but that when he chooses the right inputs, the attacker triggers the bug whatever the values of the program inputs he does not control. Secondly, we refine robust reachability quantitatively as the proportion of uncontrolled inputs that let the optimal attacker trigger the bug. We adapt symbolic execution to prove robust reachability and compute this proportion. Robust reachability is more coarse-grained because it is "all-or-nothing" but scales better than the quantitative approach. We illustrate in case studies the potential applications of these techniques, notably in terms of vulnerability assessment.

Keywords: robustness, vulnerability assessment, security, model counting, symbolic execution

---

Atteignabilité robuste et comptage de modèles pour la sécurité logicielle

Résumé

Les techniques modernes de recherche de bugs sont devenues si efficaces que le problème n'est plus de trouver des bugs mais de trouver le temps de les corriger. Une façon répandue d'éluder ce problème est de concentrer l'effort de correction de bug prioritairement sur les bugs ayant un impact en termes de sécurité, aussi désignés sous le nom de vulnérabilités. Cela conduit naturellement à la question de l'évaluation de cet impact : un attaquant pourrait-il tirer parti de tel ou tel bug ? Cette thèse se concentre sur une dimension particulière de ce problème : un attaquant serait-il capable de déclencher ce bug ? Nous appelons cette propriété la réplicabilité. Nous nous fixons pour objectif de concevoir des méthodes de recherches de bugs qui ne détectent que des bugs suffisamment réplicables. Le point de départ est de considérer des modèles de menace où l'on distingue les entrées du programme qui peuvent être choisies par l'attaquant (comme les entrées réseau) de celle qui ne peuvent ni être contrôlées ni connues de lui (comme les sources d'entropie). Nous proposons deux approches pour évaluer la réplicabilité. D'abord, nous définissons l'atteignabilité robuste, une propriété qualitative qui exprime qu'un bug est non seulement atteignable mais que lorsqu'il choisit les entrées qu'il peut correctement, l'attaquant déclenche toujours le bug, indépendamment des valeurs des entrées qu'il ne peut pas choisir. Dans un second temps, nous affinons cette approche en une approche quantitative où nous déterminons la proportion d'entrées non contrôlées qui permettent à l'attaquant optimal de déclencher le bug. Nous adaptons ensuite l'exécution symbolique pour prouver l'atteignabilité robuste ou bien calculer cette proportion. L'atteignabilité robuste est une analyse moins fine que cette approche quantitative parce qu'elle est "tout ou rien", mais en contrepartie elle passe mieux à l'échelle. Enfin nous illustrons dans des études de cas les applications potentielles de ces concepts, en particulier à l'évaluation de vulnérabilité.

Mots clés : robustesse, évaluation de vulnerabilité, sécurité, comptage de modèles, exécution symbolique

---

CEA List/LSL
CEA SACLAY Nano-INNOV – Institut Carnot LIST – DILS/LSL, Point courrier n°174 – 91191 Gif-sur-Yvette CEDEX – France

Je passe le plus rapidement possible sur trois ans de recherches, qui n'intéressent que les spécialistes, et sur l'élaboration d'une méthode de délire qui n'intéresserait que les insensés.

Marguerite Yourcenar

Remerciements

Je voudrais tout d'abord remercier mes encadrants Sébastien et Sylvain pour le temps qu'ils m'ont consacré pendant trois ans. Merci à Sébastien notamment pour tous ces moments où je ne savais plus trop dans quelle direction aller et tu m'as donné une direction claire, pour m'avoir montré la valeur de certains résultats qui me paraissaient bien insignifiants, pour ces nombreuses relectures. Merci à Sylvain pour m'avoir toujours prêté une oreille attentive, notamment à certains moments décisifs.

Merci aux membres du jury d'avoir pris le temps d'évaluer mon travail et merci tout particulièrement aux rapporteurs d'avoir relu ce manuscrit, malgré certains imprévus et délais dans la procédure administrative.

Je voudrais aussi remercier les mainteneurs de certains logiciels que j'ai utilisés pour les aspects expérimentaux de cette thèse, à qui j'ai adressé des questions, des rapports de bugs, et parfois des correctifs, et qui ont toujours pris le temps d'y répondre, et même parfois de réécrire le correctif dans son entier. Je pense notamment à Nikolaj Bjørner pour Z3, Armin Biere et Mathias Preiner pour Boolector, Christian Muise pour Dsharp, Jean-Marie Lagniez pour D4, Nian-Ze Lee pour SsatABC et Stephen Majercik pour DC-SSAT. Et, sur un autre plan, Frédéric pour Binsec, bien sûr !

Merci aux merveilleux collègues du LSL. Vous me manquerez, ainsi que les discussions dédiées à dire du mal du C, les excursions de Bühler Voyages et la brioche du monde d'avant.

Et enfin, merci à Léo de m'avoir supporté pendant le premier confinement.

Contents

Chapter 1

Introduction

1.1 Context

Software is now everywhere. Many activities are becoming increasingly reliant on computers and the software they run: shops, banks, power grids and even hospitals would now have trouble operating correctly if all this infrastructure were to cease functioning all of a sudden. A modern plane can run programs comprising several millions of lines of code, and what if one of them is wrong? What if there is a *bug*?

Correctness This anguishing question can be formulated as the issue of *correctness*: will the software running on this computer behave *as intended* by the programmer? The entire field of program verification is dedicated to this question, and decades of research have brought us numerous distinct techniques to prove that a program effectively does what it is meant to. Program verification can boast indisputable successes notably in aeronautics and public transportation [86, 2, 16, 13, 31]. Still, proving the correctness of a program is not for the faint of the heart: the program must be designed with the proof in mind, and the amount of work needed is orders of magnitudes higher than standard development, even for experts [2]. Industrially used provably correct software is more the exception than the norm.

The vast majority of software is developed with lower expectations, using methods that we regroup under the term of *bug-finding*. The main idea is to automatically search for bugs for some amount of time, in what could be described as a good faith attempt at showcasing correctness. Let us mention a few notable examples of such techniques. The most simple one is *testing*, which runs the software on inputs written by hand by a programmer, and checks that the output matches a programmer-provided ground truth. Exploration of program behaviors is limited to the cases the programmer thought of. *Fuzzing* [93] alleviates this limitation by automatically generating millions of random inputs. It then checks only that the software does not crash, so the precise behavior of the program is not actually checked. The force of this technique is that for most simple implementations it considers the program as a black box, and does not need expensive static analysis. To the contrary, *symbolic execution* [64, 22] uses such static analysis to automatically compute interesting inputs that lead to some classes of bugs, but at a far higher computational cost.

Most of these techniques have a number of characteristics in common. The number of program behaviors they can explore and check is usually finite, meaning that some bugs may be missed

"by design". Fuzzing and symbolic execution can be rephrased in terms of reachability: they attempt to answer the question to whether the program can reach a specific state (a crash for fuzzing, anything for symbolic execution). Very often in fuzzing, one reduces more complex properties like memory safety to the reachability of crashes using a monitor (like ASAN [110], Valgrind [95] or E-ACSL [115]) that checks during execution that the property is satisfied and "reaches a crash" if not.

These techniques may not replace a proper correction proof that bugs are not reachable, but their restricted scope has been a recipe for success: innumerable bugs in widely used software have been found by bug-finding techniques [111, 64, 23]. Notably, fuzzing is now becoming very widespread in industrial settings, and despite its apparent lack of subtlety, this technique is actually finding too many bugs [78]. As it is fully automatic, it can run unattended and report its results as it find bugs. Surprisingly, the bottleneck is now not *finding the bugs* but *making something of the bug reports*, a chore also known as bug triage.

Ideally, every time we find a bug, we should fix it: modify the program until for this specific input, in this specific situation, it now works correctly. But software authors usually have finite time or budget, and would rather employ it to do something else, like adding features—and further bugs—to the code, rather than fixing bugs. One aspect of bug triage is thus selecting what bugs we have the time to fix, and what others will be left for future work.

Security Let us step back for a moment. We are now dependent on software in many of our activities, even the more critical ones. An important consequence is that computer systems are also becoming increasingly valuable targets for attacks. To name only one example, even during Covid, hospitals where hit by ransomware attacks [39]. In a very abstract way, we can define an attack as the fact that an attacker attempts to use the program for his own purposes, rather than the ones the developer had in mind. This very generic formulation does not specify the goals the attacker is pursuing nor the means he uses to achieve them. This is specified in a *threat model*. A simple lock provides security against theft when your threat model considers only curious, opportunistic people, while a safe is more indicated against more powerful attackers.

In the case of computers programs, attacks can be performed by several vectors. The program can behave *as we intended*, but what we intended was wrong. In some way, this is the case of spam and phishing: the protocol of email is designed in such a way that anyone can send email with any address email in the *From:* field. This design error is at the level of the specification of the program and the program respects it. Protocol verification has seen a lot of research, and has seen successes to the point that some ubiquitous protocols like TLS1.3 were written taking into account the feedback of researchers in this field [37].

But sometimes the bug stems from a mismatch between what the programmer meant and what the program actually does. Consider the case of the Web. The browser of the user makes a request to a server. To answer this, the server is effectively running code at the request of the client. In other words, this allows the attacker (who is also a client, after all) to *run code on the server*. It is thus crucial that only restricted kinds of operations can be run on the server by the attacker: for example "reading the content of the home page and sending it back" but not "reading the password of another user and sending it back". Heartbleed [38] is a well known bug which affected a large part of web servers between 2012 and 2014. To check that his connection to the server is still alive, the attacker is allowed to request a "heart beat", by sending a piece of text t along with its length n to the server. The server is supposed to send it back. The affected software, called OpenSSL, copies n bytes of t to a temporary location in memory and then sends that back. If t is "foo" but n is 30000, OpenSSL will copy and then send 30000 bytes of memory back to the attacker. This memory will start with "foo", but the rest of the 30000 bytes may contain sensitive information, including keys or passwords. In other words, the attacker used a

bug in the program to leak secrets.

An attack can have various consequences: crash a service (Denial of Service, DoS), leak information, or even let the attacker take total control of the program. Their impact is quite context-dependent. For example, DoS is not always critical: your text processor occasionally crashing is not much more than a mere annoyance, but it becomes much more concerning with a power grid. Similarly, not all attackers can perform all attacks. Some of them can be performed remotely—the attacker can exploit them from his sofa—while others require physical access to the victim computer.

All this leads to the problem of *vulnerability assessment*: given a bug and a threat model, can the bug be used by an attacker? Only a powerful one, or any script kiddie? Could it have dire consequences if used maliciously, or only mild annoyances? This question—about *security* more than *correctness*—is important in its own right. For example, the industry settled on some commonly accepted good practices: security-relevant bugs (commonly called *vulnerabilities*) must be reported privately, and kept secret until the software can be fixed. Software vendors commonly provide security updates, which are updates which will not change the user-facing behavior but only fix security-relevant bugs. And most importantly, vulnerabilities must but be fixed quickly: vulnerabilities found by attackers before software vendors are called 0-days, meaning that software vendors have 0 days to fix the issue.

But as software vendors increasingly focus on security vulnerabilities, a parallel phenomenon can be observed: bugs without security impact are dismissed as not interesting and in some cases not fixed at all. See for example the bug referred to as CVE-2019-16230 [21]: if the system runs out of memory very early during boot, the Linux kernel might perform invalid memory accesses, and crash. This is arguably an unrealistic situation, and while it is easy to fix, the issue is not solved more than a year after the report. This illustrates what we call in this thesis a *false positive in practice*: the bug is technically valid, albeit not practical nor security-relevant, and thus not worth the time fixing.

1.2 Goal and challenges

As said before, it has become quite common to use bug-finding techniques to improve the quality of the software we depend on, but they are so successful at finding bugs that they find more than we can fix in a timely manner. *Bug triage* is the process of choosing which bugs we will fix first. Given that some bugs are a security concern while others are unlikely to be exploited by an attacker, it is quite natural to wish for a bug triage method which prioritizes vulnerabilities: the more security impact the bug has, the higher its priority. Said otherwise, we want a form of bug-finding without the false positives in practice.

This is an important shift, especially for the formal foundations of program analysis: a reachable bug is not good enough *per se* anymore, and we must shift our focus from correctness and safety to security.

There are many ways to define what constitutes a vulnerability; in some sense, there is one per threat model. We propose in this thesis to focus on one specific criterion to give a hint about how concerning a bug is: whether the bug happens in conditions that the attacker can produce. For example, a bug that can only happen when an integer in uninitialized memory at address `0xdeadbeef` has the only right value out of 2^{32} is arguably less concerning than a bug that an attacker can trigger by sending a carefully crafted packet through the network: in the former case the attacker is not able to choose the content of uninitialized memory, whereas in the latter case he can send any packet he wants and thus can trigger the bug 100% of the time. This dimension, which we call *replicability*, is of course only one of the factors that contribute to the

overall security impact to be assessed, but as we will see in case studies it is already a valuable asset. With this in mind, we will devise bug-finding techniques that only considers *replicable bugs* and filter out those which an attacker could hardly trigger, to help defenders focus on the most concerning bugs first.

Security is the original reason we are interested in replicability. However, we will see along the way that it can be of interest in other cases, notably when dealing with test suites: will this test always succeed or does its outcome depend on unexpected environmental details? Non-deterministic test suites are the bane of countless developers, and our work on replicability can be repurposed to fix them.

1.2.1 Challenges

Designing this kind of security-minded bug-finding is quite challenging, by several aspects.

Performance vs precision Any kind of semantic analysis of the behavior of programs is undecidable: it cannot be both precise, automated, and terminate. Each technique establishes its own trade-off in this regard. Fuzzing, arguably the most widespread today, insists on automation, efficiency, parallelism, and satisfies itself with rather shallow analysis and incomplete results. As we want not only bugs (crashes) but bugs which satisfy some specific property (replicability) we will need some more static analysis than what text-book fuzzing uses, by resorting to symbolic execution. This will trade efficiency for precision. As no trade-off is universally better, we will actually present two techniques, one more precise and one cheaper.

This compromise is all the more important as we will rely on particularly expensive tools like *quantified SMT solvers* [48, 11] or *model counters* [42, 94, 82], which have yet to reach the degree of maturity of unquantified *SAT* [3, 15] and *SMT solvers* [97, 96] usually used in bug-finding.

Binary level analysis Whether a bug can be exploited often depends on implementation details which are not described in the source code. The Heartbleed vulnerability mentioned above happens in code in the C programming language. What the C specification says about the actual behavior of Heartbleed is “anything can happen”. This does not help us with our analysis; it is actually meant to allow compilers to optimize in more creative ways. To know what will really happen when this code is run, we need to know what the compiler did, and thus analyze *compiled code, at the binary level*. This is quite challenging, as binary code has lost much of the structure that makes analyzing source code comparatively easier [7].

Hyperproperties As said above, bug-finding techniques can often be formulated in terms of reachability of a bug. Even for more complex properties, one usually adds instrumentation like sanitizers to reduce the problem to reachability. This is because reachability is a well-studied and comfortable setting: it can be proved by observing a single execution of the program reaching the desired location. Unfortunately, this is not the case of replicability: only observing one execution of the bug only proves the existence of the bug, but it might be a very unusual situation, which an attacker could not trigger. To give a formal underpinning to replicability, we will have to resort to hyperproperties [32], *i.e.* properties which can only be proved after observing several executions. Hyperproperties are studied since less long, and proof methods are considerably more expensive. As we will see, the most prominent one, self-composition [12], cannot even be applied to our case.

1.2.2 Proposal

We consider a deterministic program where all inputs are made explicit: to model a non-deterministic program, it is enough to add a random input modeling all sources of entropy. As with any security related work, we first need to state our threat model. Our approach consist in splitting inputs into two parts: controlled inputs, which the attacker can choose, and uncontrolled inputs, which may or may not prevent the attacker from achieving his goal. We consider threat models where the attacker first submits the controlled inputs of his choice, then the environment implicitly chooses the uncontrolled inputs. This is designed to mimic situations where the attacker sends a malicious file or network packet to attempt privilege escalation for example. We deliberately exclude interactive settings where the attacker can choose a third part of inputs depending on uncontrolled input and/or program outputs. This limitation is there to keep proof methods tractable.

We can then interpret replicability as the fact that when choosing the optimal controlled inputs, the attacker will achieve his goal for all, or for many values of uncontrolled inputs. We present two approaches derived from this foundation: one qualitative, and one quantitative.

Qualitative approach: robust reachability We refine the standard property of reachability into *robust reachability*: a (buggy) program location is robustly reachable if the attacker can choose a controlled input such that the bug is reached for all values of uncontrolled inputs. Informally, a bug which is robustly reachable is 100% replicable for the attacker, which constitutes a strong hint of its security impact.

The goal is then to design a method returning only robustly reachable bugs, but proving robust reachability is harder than standard reachability. While existing formalisms can express robust reachability, they are quite general and there are no efficient, automated proof techniques at a low-enough abstraction level to handle binary analysis. Therefore, we adapt two techniques originally designed with standard reachability in mind: symbolic execution and bounded model checking. These are more expensive than some of the more widely spread bug-finding techniques like fuzzing, but by not considering the program to be analyzed as a black box, they allow the more fined-grained understanding of program behavior we need. More precisely, these two techniques have in common that they convert the program as a symbolic set of constraints linking inputs and outputs called a formula, and then attempt to solve the formula. Our technique is based on solving a universally quantified version of this formula, and relies on the recent advances of solvers in the handling of universal quantifiers. This will of course come at a price in terms of performance and completeness.

Quantitative approach The previous method does indeed get rid of false positives in practice, but it is a bit overzealous: it equally dismisses bugs which an attacker could reproduce only in 99% of cases and bugs which only happen in 0.001% of cases at best. Intuitively, the former is more concerning than the latter. To give a formal meaning to this number of 99% or 0.001%, which we call quantitative robustness, we resort to a problem in the family of model counting called f**-E-MAJSAT**. Because of the asymmetry between controlled inputs and uncontrolled inputs, it is distinct from standard model counting, which has benefited from more research effort. The existing f**-E-MAJSAT** solving techniques come from other domains (probabilistic planning and Bayesian network inference notably) and as we will see, some techniques introduced to improve over more simple ones in these domains actually prove counter-productive for our intended application domain. As exact f**-E-MAJSAT** solving appears hard, we introduce a parametric solving algorithm to solve f**-E-MAJSAT** approximately: extreme values of the parameter reduce to existing techniques, but intermediate values enable reaching interesting

trade-offs in terms of performance and precision. Overall, this can help sort bugs in order of importance, even when all robust bugs have been fixed, for example. It pushes the trade-off further in the direction of precision at the expense of performance compared to the qualitative approach based on robust reachability.

1.3 Contributions

In this thesis we tackle the problem of designing a form of *security-minded* bug finding, which can avoid reporting bugs with lesser security impact. We assess the threat of a bug by how easily an attacker can trigger it, both qualitatively by introducing *robust reachability*, and quantitatively with *quantitative robustness*.

1.3.1 Primary contributions

We claim the following contributions

- We formally introduce the concept of robust reachability and motivate its use, showing concrete situations where it can avoid false positives in practice. We characterize robust reachability in terms of temporal logics and hyperproperties, and compare it to non-interference, thus justifying the introduction of a new formalism,

- We revisit symbolic execution and bounded model checking and show how they can be lifted to the robust case, and discuss how to adapt some standard optimizations of symbolic execution along the way. We implement and evaluate these, and show how robust symbolic execution can be used for criticality assessment of 5 existing vulnerabilities (CVEs), and compare it to standard symbolic execution. Robust symbolic execution appears to be tractable in practice, with reasonable overhead, yielding a false-positive-free reasoning,

- We introduce the notion of quantitative robustness as a quantitative extension of robust reachability. We show how to modify robust symbolic execution to enumerate paths which an attacker can trigger easily but not necessarily robustly. We illustrate in two case studies how this can be used for vulnerability assessment,

- As quantitative robustness computation relies on solving f**-E-MAJSAT** problems, we compare the effectiveness of various techniques originally developed for other kind of formulas, obtaining counter-intuitive results. As exact techniques are not as effective as hoped, we introduce a new parametric approximate algorithm yielding an interesting trade-off in terms of precision and performance.

1.3.2 Secondary contributions

Tools The software used for the experimental evidence supporting the first technical chapter of this thesis is already available as open-source. Namely, BINSEC/RSE, our robust symbolic execution engine is available at `https://github.com/binsec/cav2021-artifacts` along with the benchmark we used. As it is a work on replicability, great care has been taken for these experiments to be reproducible, notably by relying on the Nix package manager [50].

Papers The work presented in Chapter 4 has been published in "Not All Bugs Are Created Equal, But Robust Reachability Can Tell the Difference". Guillaume Girol, Benjamin Farinier, and Sébastien Bardin. In: *Computer Aided Verification*. Lecture Notes in Computer Science (2021). Springer. It has been presented at the 33$^{\text{rd}}$ International Conference on Computer Aided Verification and the 15$^{\text{th}}$ International Conference on Reachability Problems. This paper has been selected for an extension in a special issue of the journal for Formal Methods in Systems Design, which is still under review.

1.4 Outline

This thesis first features a motivational example in Chapter 2. It presents a program with bugs with varying degrees of replicability. The analysis of these bugs with standard reachability, robust reachability and quantitative robustness illustrates the potential of these techniques for vulnerability analysis.

Chapter 3 then presents some background concepts. A very generic presentation of program analysis is done, and some trade-offs about soundness, completeness and termination are discussed. We define the central property of reachability. As many of the techniques we are interested in are based on some kind of solver, we present the problem of satisfiability, especially modulo theory, and the related problem of model counting.

The main contributions of this thesis are presented in the two following chapters.

Chapter 4 presents the concept of robust reachability, a new property, stronger than standard reachability, designed to filter-out bugs that an attacker would have trouble reproducing in bug-finding techniques. It exposes how to adapt symbolic execution and bounded model checking to prove robust reachability. It features several case studies on real vulnerabilities showing practical applications of robust reachability to vulnerability assessment.

Chapter 5 presents a quantitative extension of robust reachability called quantitative robustness. As it reduces to a counting problem called f**-E-MAJSAT**, it proceeds with a presentation of existing approaches to solve f**-E-MAJSAT**. It complements them with a new, parametric, approximate algorithm based on relaxation. It presents an experimental comparison of these algorithms showing that relaxation allows to improve the trade-off between performance and precision. It finally comes back to the original problem of quantitative robustness by showing in case studies that a quantitative approach can bring finer information for vulnerability analysis, and that the better f**-E-MAJSAT** algorithms we compared can solve the instances generated for real programs.

Finally, we conclude and give some perspectives for future work in Chapter 6.

Chapter 2

Motivation

In this chapter we illustrate how it can be desirable to distinguish bugs which are replicable by an attacker from bugs which are not, and how usual bug finding techniques based on reachability come short in this respect.

A network server Figure 2.1 shows the code of a network server implementing a simplistic protocol mimicking the Heartbleed vulnerability [38]. When the server receives a network packet—potentially from the attacker—function `handle_packet` (l. 34) is called with a structure `packet` describing its length as reported by the network card, and its content. The protocol specifies that packets should start with 2 bytes specifying the type 1, 2, or 3 of request (field `type` of `struct packet_header`, l. 2). Function `handle_packet` computes a pointer to a function knowing how to handle each type of request (l. 41), and calls it with the packet (l. 53). We are interested in type 1 packets, heartbeat requests. Such a packet specifies a text that the server should send back to the client, along with its length (see `struct heartbeat_packet_header`, l. 6). The handler function `handle_heartbeat` (l. 22) first allocates a buffer for the answer packet with `malloc` and then copies the text of the heartbeat request into this buffer with `memcpy`.

Three more or less subtle bugs were inserted in this code:

Bug 1 when a packet has neither type 1, 2, nor 3, the function pointer `handler` is `NULL` when called l. 53, leading to a crash;

Bug 2 line 28, `memcpy` copies an attacker-controlled numbers of bytes of memory from a possibly short buffer, leaking memory to the network. This is our reimplementation of Heartbleed.

Bug 3 line 28, `memcpy` writes to a pointer allocated with `malloc` which may be `NULL` if the system runs out of memory. This can lead to a crash.

Vulnerability assessment These three bugs do not have the same security impact. Let us first compare **Bug 1** and **Bug 3**, because they are more comparable. When they happen on a system with memory protection, they should both result in a crash (because a null pointer is executed or written to): this is a case of denial of service without possibility of further exploitation. Vulnerability assessment thus somehow reduces to how easily an attacker can trigger these bugs. To trigger **Bug 1**, it is enough to send a network packet with a `type` field set to a value other than 1, 2, or 3. On the contrary **Bug 3** only happens when the system runs out of memory, which depends on other applications running on the system, and which an attacker does neither control

```
1  struct packet_header {
2    unsigned short type; // 1 for heartbeat requests, 2 or 3 for other things
3    char data[]; // actual content depends on the request type
4  };
5
6  struct heartbeat_packet_header {
7    unsigned short type; // always 1
8    unsigned short heartbeat_len; // the length of the string to send back
9    char hearbeat_request[]; // the string to send back
10 };
11
12 struct packet {
13   unsigned short len; // the length of the packet
14   struct packet_header *buffer; // the content of the packet
15 };
16
17 struct heartbeat_packet {
18   unsigned short len;
19   struct heartbeat_packet_header *buffer;
20 };
21
22 struct packet handle_heartbeat(struct heartbeat_packet packet) {
23   struct packet answer;
24   answer.len = packet.buffer->heartbeat_len;
25   answer.buffer = malloc(answer.len);
26   // does not check if malloc failed
27   // might copy more bytes than the incoming packet size
28   memcpy(answer.buffer, &packet.buffer->hearbeat_request, answer.len);
29   return answer;
30 }
31
32 typedef struct packet (*handler_t)(struct packet);
33
34 void handle_packet(struct packet packet) {
35   if (packet.len <= 4) {
36     // packet is too small for the headers we expect
37     return;
38   }
39   // the function that implements this type of request
40   handler_t handler = NULL;
41   switch (packet.buffer->type) {
42     case 1:
43       handler = (handler_t)&handle_heartbeat;
44       break;
45     case 2:
46       handler = &handle_something_else;
47       break;
48     case 3:
49       handler = &handle_irrelevant;
50       break;
51     // forgot the default case
52   }
53   struct packet answer = handler(packet);
54   send(answer);
55 }
```

Figure 2.1: A vulnerable network server

Input	**Bug 1**	**Bug 2**	**Bug 3**
`esp`	`0xb0000000`	`0xb0000000`	`0xb0000000`
packet length	16	14	16
packet address	`0x10000000`	`0x10000000`	`0x10000000`
packet	`0x0000000000000000`	`0x0000000000100001`	`0x0000000000000001`
return value of `malloc`	—	`0x20000000`	0

Table 2.1: Inputs reported by symbolic execution which trigger the bugs

nor even know about. The allocation of line 25 is limited to a short integer, thus less than 65 kB, which is unlikely to exhaust memory on modern hardware. In other words, informal analysis suggests **Bug 3** only happens in conditions which are both unlikely and out of the control of the attacker. We thus conclude that **Bug 1** is more severe than **Bug 3**.

Bug 2 is less comparable to the other two as it does not result in denial of service, but in an information leak. In some sense which outcome is worse depends on the threat model. However, note that how *replicable* the bug is is still an information of paramount importance: if, like **Bug 3**, **Bug 2** only happens in unlikely cases which are out of the control of the attacker, it seems more legitimate to defer fixing it than if it is 100% replicable by an attacker like **Bug 1**. **Bug 2** happens whenever the attacker sends a packet of type 1 with a large field `heartbeat_len` and `hearbeat_request` empty, for example, and when the allocation of line 25 succeeds. In other words, when the attacker acts optimally, he has a high, but not 100%, chance of triggering **Bug 2**.

The problem with standard reachability Assume for example, that we only have the time budget to fix two bugs. We would like to design a bug-finding technique that detects **Bug 1** and **Bug 2** but overlooks **Bug 3**, as it only happens in corner cases. Let us analyze what happens with a standard technique, symbolic execution. All three bugs are reported as reachable by the symbolic execution engine of Binsec: this technique is not able to distinguish between high- and low-replicability bugs.

This technique additionally reports one input triggering each bug (presented in Table 2.1): could we reason on them to reproduce the vulnerability assessment of the previous paragraph? As expected, one can see that **Bug 3** is triggered when `malloc` returns 0, however nothing excludes that it would work with another value. This report is compatible with the bug happening all the time instead of only in corner cases. Conversely, **Bug 1** is reported to work when the starting address of the stack `esp` is `0xb0000000`, and while we expect it to also work with many other addresses (due to Address Space Layout Randomization (ASLR) for example), nothing in this report guarantees it. This bug could only work for this specific value of the starting address of the stack `esp`, and since the attacker cannot choose this starting address, it would make it hard for him to trigger the bug.

This lack of information on the behavior of the program for other inputs is not a problem for all inputs. If all inputs were attacker-chosen, a report like the ones of Table 2.1 would be enough to conclude to the possibility of an attack, as the attacker could choose the right inputs. The problem actually arises when considering inputs that the attacker cannot control, like `esp` and the return value of `malloc`, which we mentioned above.

The fact that these reports are not so useful to distinguish high- and low-replicability bugs and more generally for vulnerability assessment is not specific to symbolic execution or its im-

plementation in BINSEC: it affects all techniques based on reachability. Reachability only tells us that there is one input leading to the bug, leaving the possibility that it is the single one, and that the attacker needs to be lucky enough for the inputs he cannot set exactly to the values in the report to spontaneously take the right value. For this reason we will propose new, stronger concepts, that express that bugs are not only reachable, but replicable for an attacker.

First proposal: robust reachability We explicitly partition the inputs of the program into two parts: *controlled inputs*, that are chosen by the attacker, and *uncontrolled inputs* which the attacker cannot influence and cannot predict. In our case, controlled inputs are the packet content and length, while the initial state of the memory and registers is uncontrolled. In particular, the starting address of the stack `esp` and the return value of `malloc` is uncontrolled. A bug is said to be robustly reachable if the attacker can choose controlled inputs such that the bug is guaranteed to be triggered for all values of uncontrolled inputs.

Bug 1 is robustly reachable because when the attacker chooses a packet of length 16 and containing only zeroes the bug is actually guaranteed to happen, for all values of `esp` and the packet addresses. This proves that this bug is replicable for an attacker. This analysis, which we obtain mechanically with symbolic execution in 0.1s, is much more precise than what we obtained with standard reachability as it enables us to conclude that this bug has the potential to turn into a high-profile denial of service attack.

Limitations The definition of robust reachability embeds two main intentional limitations.

Firstly, when we partition inputs into controlled inputs which are attacker chosen and uncontrolled inputs which are unknown to the attacker and cannot be influenced by him, we implicitly forbid interactive systems where the environment first chooses some uncontrolled input x_1, then the attacker, given the knowledge of x_1, chooses a first controlled input a_1, then in reaction the environment chooses another uncontrolled input x_2 which may prevent the attacker from achieving his goal, then the attacker may be allowed one second input a_2 to the system, *etc.* The reason of this design choice is that we want to take advantage of the abilities of Satisfiability Modulo Theory (SMT) solvers to solve quantified formulas. Such interactivity would require numerous quantifier alternations: $\forall x_1.\, \exists a_1.\, \forall x_2.\, \exists a_2.\, \mathsf{bug}$ where models for existential parameters are actually functions, which is very badly supported by state-of-the art solvers (see also Section 4.6.4.3). Our non-interactive definition of robust reachability only requires one quantifier alternation: $\exists \mathsf{controlled}.\, \forall \mathsf{uncontrolled}.\, \mathsf{bug}$ which appears reasonably tractable in our experiments.

The second limitation is visible when we try to apply the definition of robust reachability to the two other bugs. **Bug 3** is not robustly reachable because it only happens for some values of uncontrolled inputs (when `malloc` returns 0). We can thus conclude that this bug is not replicable all the time by an attacker, but not that it only happens rarely. **Bug 2**, which we manually analyzed to happen all the time except in the unlikely case where memory allocation fails, is not robustly reachable because even for the right controlled inputs, the bug is not triggered successfully for all values of uncontrolled inputs: it fails to happen when `malloc` returns 0. Said informally, robust reachability only detects bugs which are 100% replicable and dismisses bugs which are replicable only 99% of the time. As a result, **Bug 2** and **Bug 3** are indistinguishable in terms of robust reachability, whereas we expect **Bug 2** to be much more replicable. Robust reachability has been designed to be "all-or-nothing" to be able to rely on the theory of universal quantification in SMT solvers, but we would like something more quantitative that can accommodates bugs which are, like **Bug 2**, replicable "often, but not always". This leads us to a second proposal relying not on universal quantification but on model counting, which is expected to be more expensive but more precise.

Second proposal: a more quantitative approach We now try to assess quantitatively how replicable a bug is in the following sense: for the optimal controlled input, we compute the proportion of uncontrolled inputs that trigger the bug. We call this maximal proportion quantitative robustness. To do such model counting, we need to specify the domain of all inputs. Notably, we model the return value of `malloc` as an integer which is either 0 or in a heap located at [`0x20000000`, `0x40000000`]. In these conditions, one can compute mechanically in 2 seconds that **Bug 3** only happens for $\frac{1}{2^{29}+2}$ of uncontrolled inputs at best (when malloc returns 0) even when the attacker chooses controlled input optimally. On the other hand, one can compute in about 2 seconds that **Bug 2** happens with quantitative robustness above 0.999999998137 (exact solving times out). This value is compatible with the expected one: $1 - \frac{1}{2^{29}+2}$. This illustrates how quantitative robustness allows distinguishing automatically these two bugs: **Bug 2** is more replicable than **Bug 3**.

Quantitative robustness also has the advantage that it is compatible with the previous notions: a bug is unreachable when it has quantitative robustness 0 and robustly reachable when it has quantitative robustness 1. With this methodology, we compute that **Bug 1** has quantitative robustness 1 and can compare its replicability successfully to those of **Bug 3** and **Bug 2**.

Scope Note that we compute a proportion of inputs triggering a bug, not a probability. If a probability is desired, our method can be interpreted as allowing only a uniform probability distribution for uncontrolled inputs. One could want to specify an arbitrary probability distribution, but as we will see effective techniques to compute quantitative robustness are restricted to some kind of distributions only. But beyond the capabilities of the technique it is good to think about the capabilities of the end user. Does the user actually know the probability distribution of the return value of `malloc`? In practice, it is not the case, so it would hardly be a gain to extend our tool to allow specifying such a custom distribution. Besides, our goal here is to distinguish between a bug that only happens when the system is out of memory and one that always works *except* when the system is out of memory. In this case the uniform distribution is enough to highlight this very strong asymmetry. This technique is thus designed to give a hint of the replicability of bugs, not necessarily to obtain an assessment of the vulnerability as a 10-digit-precise score.

It is also the right place to reiterate that we only consider replicability here: many other dimensions also matter. In our case **Bug 2** is an information leak while **Bug 1** and **Bug 3** are denial of service bugs, and therefore have a very different impact. Our techniques should be considered as tools helping the (presumably manual) process of vulnerability assessment but not replacing it.

Chapter 3

Background

In this chapter we present some basic concepts which are good to have in mind for the developments to come. Firstly, we will present ways to express the properties that a program satisfies (trace properties, hyperproperties) and how to qualify the corresponding proof methods (correctness, completeness). Secondly we will define what a solver tries to prove: satisfiability of boolean formulas, satisfiability modulo theory, and model counting.

3.1 Program analysis

Program analysis designates the task of determining whether the actual behavior of a program satisfies a property. In full generality, there are several ways to give a formal meaning of what "program behavior" and "property" means, hence the abstract phrasing. For example, one property we could like to prove is "the program never crashes". A number of techniques have been developed to solve this kind of problem, usually regrouped under the name of *formal methods* [31, 106, 35, 17, 68].

We will present the simple formalism of trace properties in transition systems, and use it to introduce useful characteristics of such methods. The formalism of trace properties is slightly too restricted for the discussions to come, so we will also introduce hyperproperties [32].

3.1.1 The object: transition systems

To describe a program P we consider the set $\mathcal{S}$ of all possible states of the machine executing it. A state can be the pair of the memory and register content of an idealized computer, or something more abstract, like the number of the next line of code to be executed and the current value of all variables. Consider the example of Figure 3.1: the initial state of the program can be

```
1 int s = 0; int i = 0;
2 for (i=1; i<70000; i++) {
3   s += i;
4 }
```

Figure 3.1: Example program

expressed as $\{\texttt{line} \mapsto 2, \texttt{s} \mapsto 0, \texttt{i} \mapsto 0\}$, where `line` denotes the next line of code to be executed, and `s` and `i` refer to the variables of the program. This example is very simple, but in general the initial state depends on a program input $y \in \mathcal{Y}$: we denote the initial state as $s_1(y)$.

The program P is represented by a (one-step) successor relation $\to \in \mathcal{S} \times \mathcal{S}$ expressing in what states the program can go on next instruction. For example the `s += i` instruction corresponds to the transitions $\{\texttt{line} \mapsto 3, \texttt{s} \mapsto 0, \texttt{i} \mapsto 1\} \to \{\texttt{line} \mapsto 2, \texttt{s} \mapsto 1, \texttt{i} \mapsto 1\}$, $\{\texttt{line} \mapsto 3, \texttt{s} \mapsto 3, \texttt{i} \mapsto 6\} \to \{\texttt{line} \mapsto 2, \texttt{s} \mapsto 9, \texttt{i} \mapsto 6\}$ and more generally $\{\texttt{line} \mapsto 3, \texttt{s} \mapsto n, \texttt{i} \mapsto m\} \to \{\texttt{line} \mapsto 2, \texttt{s} \mapsto n + m, \texttt{i} \mapsto m\}$ for all integers n, m. We denote as $\to^+$ the transitive closure of $\to$.

A *trace* t is a sequence of states respecting $\to$: for i an integer, $t_i \to t_{i+1}$ where t_i denotes the i-th state of t. An example of trace is $(\{\texttt{line} \mapsto 2, \texttt{s} \mapsto 0, \texttt{i} \mapsto 0\}, \{\texttt{line} \mapsto 3, \texttt{s} \mapsto 0, \texttt{i} \mapsto 1\}, \{\texttt{line} \mapsto 2, \texttt{s} \mapsto 1, \texttt{i} \mapsto 1\})$. If a program does not terminate, then traces can be infinite. We denote as $\mathcal{S}^+$ all finite sequences of states, as $\mathcal{S}^\omega$ all infinite such sequences, and as $\mathcal{S}^\infty \triangleq \mathcal{S}^+ \cup \mathcal{S}^\omega$ the set of all state sequences. The program P is accurately depicted as the set of traces it admits, denoted as $T(P) \subseteq \mathcal{S}^\infty$. Note that we consider partial traces here: if P admits a trace t, it also admits all its prefixes.

Later, we will define properties of the form "line 3 of the program is reachable". To model this, each state $s \in \mathcal{S}$ has a corresponding location $\lambda(s)$, for example a line nuber in the source code. We distinguish traces from paths: a path is a sequence of locations (a synctactic feature), while a trace is a sequence of states (related to the semantics of the program). A path can correspond to many traces.

3.1.2 The proof goals: trace properties

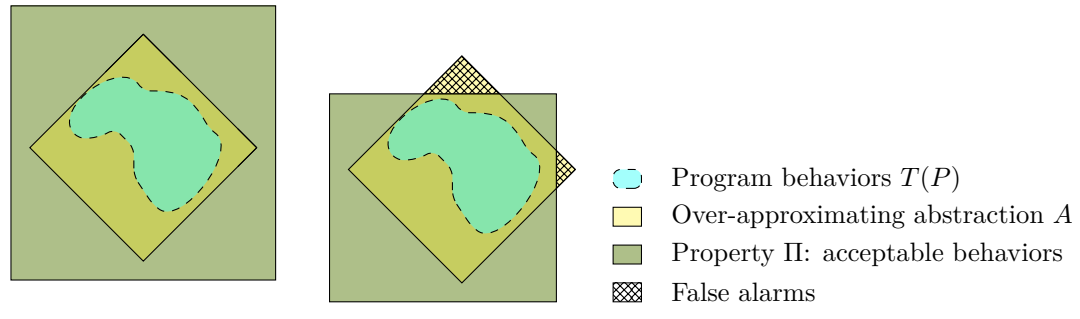
A trace property Π is the set of traces it allows: $\Pi \subseteq \mathcal{S}^\infty$. A program P satisfies the property Π, denoted as $P \vdash \Pi$, if all its traces are allowed: $T(P) \subseteq \Pi$.

A well known class of trace properties is safety properties, informally presented as "something bad must not happen", for example "crashes must not happen", or "division by 0 must not happen". In our example program Figure 3.1, one could want to prove that the variable `s` is never negative. The corresponding property is $\Pi = \{\forall s \in t.\, s[\texttt{s}] \geq 0 \mid t \in \mathcal{S}^\infty\}$. It happens that for 32-bit-wide integers, P does not satisfy Π because s overflows: at the 65637-th iteration the trace ends with $\{\texttt{line} \mapsto 3, \texttt{s} \mapsto 2154009430, \texttt{i} \mapsto 65637\}, \{\texttt{line} \mapsto 2, \texttt{s} \mapsto -2140892230, \texttt{i} \mapsto 65637\}$. This trace does not belong to Π and thus $P \nvdash \Pi$.

Another kind of property expressible as trace property is termination. Termination is the property $\Pi = \mathcal{S}^+$. Our example program does terminate: $T(P) \subseteq \mathcal{S}^+$.

3.1.3 Reasoning by abstraction

The field of verification could be summarized as follows: for a given property Π, we consider the decision problem of determining for any program P if $P \vdash \Pi$. Rice's theorem [105] implies that if Π is not the always false property $\varnothing$ nor the always true property $\mathcal{S}^\infty$, this decision problem is undecidable. As a consequence, verification cannot be both precise, automatic, terminating and generic (working on all programs). Various techniques for program verification forgo one or several of these qualifiers. *Deductive verification* [68] is not fully automatic as it relies on a human operator to provide annotations like pre-, post-conditions and invariants, or even write full proofs. *Model checking* [17] models the program to be verified as abstracted finite system, and verifies the finite system. Finiteness allows to eschew undecidability. *Symbolic execution* [22] needs to enumerate all paths in the program, which may be infinitely many, and will therefore



(a) Proving a property with over-approximation

(b) Failure to prove a true property because of false alarms

Figure 3.2: Over-approximating proof method

often not terminate. *Abstract interpretation* [35, 106] may return a wrong result in some cases, in that it raises alarms for bugs which actually do not exist.

One way to eschew undecidability is to first approximate the behavior of the program into a so-called *abstraction* A and then decide whether the abstraction satisfies Π. There are two main ways to perform this.

Over-approximation The abstraction contains all the behaviors of P, plus some extra ones. In terms of trace properties: $T(P) \subseteq A$ and we compute whether $A \subseteq \Pi$.

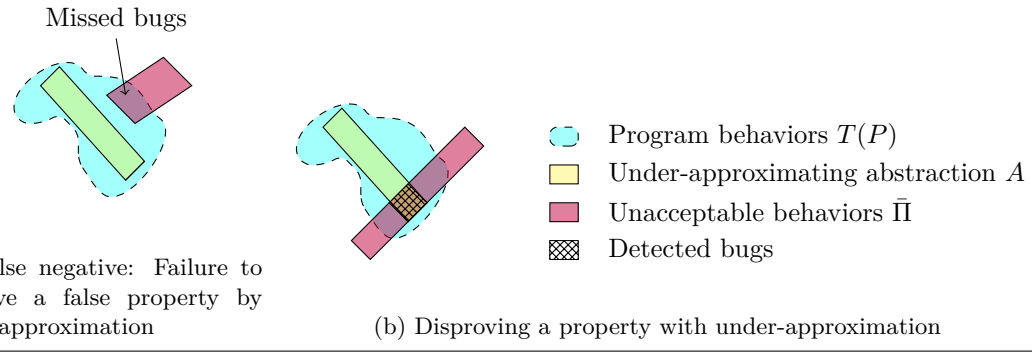
Under-approximation The approximation only admits behaviors in P, but misses some of them. In terms of trace properties: $A \subseteq T(P)$, and we compute whether $A \subseteq \Pi$. This is the case of most bug-finding techniques.

The textbook example of over-approximation is abstract interpretation [35]: instead of storing the set of all possible values of a variable (and their relations), it will approximate them into a superset of a specific form which is both concise and easy to reason about. For example, in Figure 3.1 one can compute the set of possible values of for variables `s` and `i` line 3 as an interval. For `i`, one finds $[1, 70000]$, which is perfectly precise. However, for `s` it would be $[-2140892230, 2154009430]$, a set orders of magnitude larger than the 70000 expected values.

An example of under-approximation is symbolic execution [22]. Symbolic execution works by first enumerating all paths in the program, unrolling loops. Then it computes a set of constraints (under a form described in Section 3.2) on the variables of the program expressing that the path is taken, and attempts to determine whether these constraints are satisfiable. If the path syntactically leads to a bug and the constraints leading to it are satisfiable, then we found a bug. This technique is an under-approximation because the set of paths of a program is most often infinite, and in practice one bounds exploration. Some behaviors (and possibly bugs) are thus missed.

An over-approximation can consider fictitious behaviors (traces) which violate Π whereas P satisfies it. This leads to **false positives**, or **false alarms**: a bug is found where there was none. (See Figure 3.2) Over-approximating techniques cannot have false-negatives: they cannot conclude to the absence of bugs if there are bugs, since all behaviors are included in the abstraction.

Conversely, under-approximations miss some behaviors, and thus may miss bugs: this would lead to a **false negative**. See Figure 3.3. For example, conventional testing misses all the bugs in code not exercised by the test suite. On the other hand, under-approximating techniques do not have false positives as all behaviors in A actually belong to the program P.



(a) False negative: Failure to disprove a false property by under-approximation

(b) Disproving a property with under-approximation

Figure 3.3: Under-approximating bug-finding

This opposition is often described in terms of correctness and completeness. Techniques which terminate can be modeled as an algorithm V which takes a program P as input and returns 1 as an imprecise indication that $P \vdash \Pi$, or 0 otherwise. Such verifiers can be classified into two categories:

Completeness V is complete with respect to Π if $\forall P. V(P) = 0 \implies P \nvdash \Pi$. The adjective complete is to be understood as "will find *all* violations of Π". Over-approximating techniques are normally complete, but not correct.

Correctness V is correct (or sound) if $\forall P. V(P) = 1 \implies P \vdash \Pi$. V can prove the property, but not its negation. Under-approximating techniques are normally correct, but not complete.

This can be usefully extended to non-terminating techniques. For example, symbolic execution [22] and bounded model checking [29] are under-approximations where the abstraction quality is parametrized by a bound k. They only consider traces of length at most k: $A = \{t \in P \mid |t| \leq k\}$, where $|t|$ denotes the length of trace t. These techniques are incomplete, but informally they would become complete with infinite time. One can make this more formal as follows:

k-Completeness V is k-complete with respect to property Π if for all bounds k and all programs P where traces are at most length k, $V(P, k) = 0$ implies that $P \nvdash \Pi$.

In other words, the finite restriction of the verifier to bounded programs is complete for all bounds k.

3.1.4 Beyond trace properties: hyperproperties

Reachability A property central to our later discussions is reachability. It expresses whether there is an initial state of the program can lead to an event. Many practical bug-finding targets can be reduced to reachability of bugs. Elementary examples are "reachability of division by 0" or "reachability of null pointer dereference" for example, but more complex instances also do. For example, consider memory safety. It is a complex property especially if one tries to define it formally. The most frequent way to dealing with it in fuzzing is to use a monitor like ASAN [110], Valgrind [95] or E-ACSL [115]: a program which runs the program P and computes whether the current execution trace satisfies memory safety. If a violation is detected, the monitor crashes with a helpful diagnostic. Fuzzing then only has to determine whether a crash of the monitor is reachable.

Formally, let us attempt to define reachability of line ℓ of the program. Reachability of a program location ℓ means that there is an input to the program such that execution leads to line ℓ:

$$\exists y \in \mathcal{Y}.\, \exists s \in \mathcal{S}.\, s_1(y) \rightarrow^+ s \wedge \lambda(s) = \ell$$

The framework of trace properties we described above can easily express the negation of reachability: it is the property $\Pi = \{t \in \mathcal{S}^\infty \mid \forall i.\, \lambda(t_i) \neq \ell\}$. The program must only admit traces avoiding the location ℓ. However it cannot express reachability directly. For this reason we introduce a more complex framework called hyperproperties.

Hyperproperties A hyperproperty [32] is a set of sets of traces $\Pi \in 2^{2^{\mathcal{S}^\infty}}$. A program P satisfies a hyperproperty Π if $T(P) \in \Pi$. Hyperproperties are considerably more expressive than trace properties (it is easy to see that a trace property can be converted into a hyperproperty). Notably they can express properties relating several traces, like "this program terminates on average in less than 1s". Hyperproperties are generally thought to be harder to prove than trace properties. For example some categories of hyperproperties relating two traces (2-hypersafety) can reduced to trace properties with self-composition [12] but this comes at the cost of squaring the size of the program if implemented naively.

Reachability of program location ℓ can be expressed as a hyperproperty:

$$R(\ell) = \{P \in 2^{\mathcal{S}^\infty} \mid \exists t \in P.\, \exists i.\, \lambda(t_i) = \ell\}$$

but it is not among the hardest ones, as its negation is a trace property, and a well-studied one at that.

While reachability of a program location ℓ is a simple mental model, it is a bit restricted. We can generalize it to reachability of any event given under the form of a set of finite traces $O \subseteq \mathcal{S}^+$.

Definition 1 (Reachability)**.** A set of finite traces $O \subseteq \mathcal{S}^+$ is reachable in program P if $P \cap O \neq \varnothing$. We write $P \vdash R(O)$.

As a special case one gets the usual property of reachability of a location:

Definition 2 (Reachability of a location)**.** A location is reachable, denoted as $P \vdash R(\ell)$, if $P \vdash R(O)$ with $O = \{t \in \mathcal{S}^+ \mid \lambda(t_{|t|}) = \ell\}$.

Link to the thesis In this thesis, we argue that while bug-finding techniques like symbolic execution are sound, *i.e.* that they have no false positives, in a security-oriented context they can report bugs which are wasting everyone's time because they could only happen in conditions which an attacker could not reproduce. We call them *false positives in practice*. This issue is intrinsic to reachability and not specific to a particular technique. We attempt to design new techniques which avoid false positives in practice, starting by refining reachability into a stronger property.

3.2 Satisfiability of formulas and related problems

Some techniques like symbolic execution [22] and bounded model checking [29] work by converting a path into a set of constraints over the program input. Such a set of constraint is often called a *formula*. If the program input satisfies the constraint represented by the formula, then the program will follow the original path in the program. In this section, we present some variants of formulas and problems that apply on such formulas.

3.2.1 Propositional formulas

We consider the theory of propositional formulas over a finite set of boolean variables $v \in \mathcal{V}$. The set $\mathcal{F}$ of propositional formulas is defined starting from variables $v \in \mathcal{V}$, and for $f, g \in \mathcal{F}$ adding negation $\neg f$, conjunction $f \wedge g$ and disjunction $f \vee g$. We denote as $V(f)$ the set of variables appearing effectively in a formula f. Propositional formulas are usually given in Conjunctive Normal Form (CNF). A literal is v or $\neg v$ where v is a variable. A clause is a set of literals, interpreted as their disjunction, and a formula in CNF is a set of clauses, interpreted as their conjunction.

A partial valuation is a partial mapping from a subset of $\mathcal{V}$ to the set $\mathbb{B} \triangleq \{\top, \bot\}$. One can apply a partial valuation m to a full formula f: $f|_m$ is the formula identical to f where variables v in the domain of m are replaced by $m(v)$. For example, for $f = v_1 \wedge (\neg v_1 \vee v_2)$ and $m = \{v_1 \mapsto \top\}$, the formula obtained by applying m on f is $f|_m = v_2$.

A valuation is complete for f when its domain contains $V(f)$, *i.e.* it associates all variables to a boolean value. Such a valuation maps a propositional formula to $\mathbb{B}$ as well.

A complete valuation m is said to be a model of a formula f if $f|_m = \top$ We denote as $M(f) \triangleq \{m \in \mathbb{B}^{V(f)} \mid f|_m = \top\}$ the set of models of a formula f, and as $\sharp(f) \triangleq |M(f)|$ its cardinal. For example, the models of $v_1 \wedge (v_2 \vee \neg v_2)$ are $\{v_1 \mapsto \top, v_2 \mapsto \bot\}$ and $\{v_1 \mapsto \top, v_2 \mapsto \top\}$.

Note that the definition of models depends on the number of variables of a formula. Therefore, $\sharp(v_1) = 1$ whereas $\sharp(v_1 \wedge (v_2 \vee \neg v_2)) = 2$. This will be discussed when we introduce smoothness (Definition 18). Then we can define two propositional formulas f, g as equivalent when $M(f) = M(g)$.

3.2.2 Satisfiability, model counting and related problems

We now introduce several standard problems related to boolean formulas.

Definition 3 (satisfiability)**.** SAT is the following decision problem: given propositional formula f in CNF, output whether $M(f) \neq \varnothing$.

A formula with an empty set of models is said to be unsatisfiable, and is said to be satisfiable otherwise.

Definition 4 (model counting)**.** $\sharp$SAT is the following function problem: given a propositional formula f in CNF, output $\sharp(f)$

It is well-known that SAT is NP-complete while $\sharp$SAT is $\sharp$P-complete [119]. Actually, $\sharp$SAT straightforwardly generalizes SAT as a formula is satisfiable if and only if its model count is non-zero, but it is also at least as hard as the polynomial hierarchy PH, *i.e.* the satisfiability of propositional formulas with any constant number of universal and existential quantifiers [118].

3.2.3 Satisfiability modulo theory

Boolean formulas fit the need of *e.g.* hardware verification but they turned out not to be very handy for program analysis. For example, to analyze the program of Figure 3.1, we would like to be able to write constraints over integers and to use + for the addition. In the context of formulas, + will be a new *symbol*, integers will be a new *sort* and the set of these new symbols and sorts with their governing axioms is called a theory. Depending on the application, one can need several distinct theories: we could use natural integers, or bounded machine integers with modulo arithmetic for example. With the latter we write $s_0 = 0 \wedge i_0 = 0 \wedge i_1 = 1 \wedge s_1 = s_1 + i_1 \wedge i_2 = 2 \wedge s_2 = s_1 + i_2 \wedge s_2 \leq_s 0$ to express that an overflow happens after two iterations

of the for loop. $\leq_s$ is a symbol for signed comparison. One can extend the notion of satisfiability to such formulas with theories, and conclude that since this formula is not satisfiable, there can be no overflow after 2 iterations of the loop in the program of Figure 3.1.

Definition 5 (Signature)**.** A signature is a triple $\Sigma = (S_\Sigma, F_\Sigma, P_\Sigma)$ where

- S_Σ is a set of sort symbols;
- F_Σ is a set of function symbols $f : s_1 \times \cdots \times s_n \to s$ where n is the arity of f, $(s_1, \ldots, s_n) \in S_\Sigma^n$ are the input sorts of f and $s \in S_\Sigma$ is the output sort of f;
- P_Σ is a set of predicate symbols $p : s_1 \times \cdots \times s_n$ where n is the arity of p, and $(s_1, \ldots, s_n) \in S_\Sigma^n$ are the input sorts of p.

One builds a formula in several steps: first terms from signature symbols, then a formula from predicates on terms and logical connectives.

Definition 6 (Term)**.** For Σ a signature and a set $\mathcal{V}$ of sorted variables, the set of Σ-terms is recursively defined as

- variables;
- $f(t_1, \ldots, t_n)$ for $f \in F_\Sigma$ a function symbol and t_1, ..., t_n terms of the correct sorts.

Definition 7 (Formula)**.** The set of formulas is defined as

- $\top$, $\bot$;
- usual logical connectives over formulas f, g: $f \vee g$, $f \wedge g$, $\neg f$;
- $p(t_1, \ldots, t_n)$ where $p \in P_\Sigma$ is a predicate and $t_1, \ldots, t_n$ are terms of matching sorts;
- quantifiers $\forall v.\, f$, $\exists v.\, f$.

A formula without quantifiers is said to be quantifier free. A variable never bound by a quantifier is said to be free, and a formula without free variables is said to be closed.

For example, a formula in Presburger arithmetic $\Sigma = (\{\text{Int}\}, \{0, 1, +\}, \{=\})$ is $\forall x. \forall y. (x+1 = 1+y) \implies x = y$.

Now we attempt to assign a truth value to a formula to define its satisfiability. We first need to give semantics to variables with an assignment, and to the symbols of the signature with an interpretation.

Definition 8 (Interpretation)**.** A Σ-interpretation I is a mapping from

- each sort symbol s to a set $[\![s]\!]$;
- each function symbol $f : s_1 \times \cdots \times s_n \to s$ to a function $[\![f]\!] : [\![s_1]\!] \times \cdots \times [\![s_n]\!] \to [\![s]\!]$;
- each predicate symbol $p : s_1 \times \cdots \times s_n$ to a function $[\![f]\!] : [\![s_1]\!] \times \cdots \times [\![s_n]\!] \to \mathbb{B}$.

Definition 9 (Assignment)**.** An assignment is a mapping from each free variable v of sort s in $\mathcal{V}$ to an element $[\![v]\!]$ of the set $[\![s]\!]$. This definition depends on the interpretation of sorts.

Then we can lift the pair of the interpretation and assignment $[\![\cdot]\!]$ to terms and formulas:

Definition 10 (Value of a term)**.** The value $[\![t]\!]$ of a term t is defined as follows:

- for a variable v, the value assigned $[\![v]\!]$ to it by the assignment.
- for a function symbol $f(t_1, \ldots, t_n)$, the value $[\![f]\!]([\![t_1]\!], \ldots, [\![t_n]\!])$.

Definition 11 (Truth value of a formula)**.** Under an interpretation and assignment $[\![\cdot]\!]$, the truth value of a formula is defined as follows:

- For a predicate $p(t_1, \ldots, t_n)$, $[\![p]\!]([\![t_1]\!], \ldots, [\![t_n]\!])$;
- usual logical connectives keep their usual truth table
- for v a variable of sort s, $\forall v.\, f$ is true if and only if f is true with the same interpretation, and in all assignments that coincide with $[\![\cdot]\!]$ but also assign all elements of $[\![s]\!]$ to v.

For example, again in Presburger arithmetic, $x+1 = y$ is true under the interpretation which maps Int to the usual set of natural integers $\mathbb{N}$, $+$, 0, 1, $=$ to their usual meaning, and the assignment mapping x to 2 and y to 3.

It is usually not enough that a formula is true in isolation. We also want it to satisfy a set of axioms, otherwise $x = y \wedge y = z \wedge x \neq z$ could be made true.

Definition 12 (Theory)**.** A theory is a signature along with a set of formulas over this signature called axioms.

For example, Presburger arithmetic contains axioms like commutativity of the addition: $\forall x. \forall y.\, x + y = y + x$.

Definition 13 (Satisfiability modulo theory)**.** A formula f is satisfiable if there is an interpretation and assignment that make it true along with all axioms of the theory.

In general, this problem, called SMT, is undecidable. For some specific theories, like Presburger arithmetic, satisfiability is decidable [20]. This is also the case of a number of quantifier-free theories. We are presenting one example next.

3.2.4 Bitvectors and arrays

The theory we use in the tools developed in this thesis is the theory of bitvectors and arrays.

Bitvectors Bitvectors model the finite-width integers used-by computers with 2^n modulo arithmetic. The signature of the theory contains a sort symbol BitVec(n) for all integers $n > 0$ corresponding to integers over n bits, from 0 to $2^n - 1$. Function symbols include modular arithmetic for each bitwidth: $+_n$, $-_n$, $\times_n$, euclidean division and remainder, as well as bitwise operations: negation, logical AND, OR, XOR, bitshifts. For example, in size 8, $254 +_8 3 =_8 1$. Some more bitvector-specific operations like concatenation and extraction are also provided. For example, the concatenation of 0100 and 100 (given in base 2) is 0100100. Predicates are usual comparison and equality, with the added subtlety that comparisons like $<_n$ come in signed $<_{n,s}$ and unsigned $<_{n,u}$ variety: over 8 bits, 254 can be interpreted as -2 in two-complement signed integers so $254 >_{8,u} 10$ but $254 <_{8,s} 10$.

Bitblasting Bitvector sorts have a finite domain and bitvector theory is therefore decidable. This can be seen by a technique called bitblasting which reduces a bitvector formula to a an equisatisfiable propositional formula. The idea is that for each bitvector b of size n, n propositional variables $B_1, \ldots B_n$ are introduced representing the bits of b. When bitvector terms are constructed, constraints over the bits of the result must be appended to the formula. For bitwise

operations like $c = a\&_n b$, the bits of the result verify $c_n \Leftrightarrow a_n \wedge b_n$, but for arithmetic operations like addition and multiplication, the involved constraints are considerably larger and require the introduction of auxiliary variables.

In the end, from a bitvector formula $\phi(b_1, \ldots, b_n)$ with free variables $b_1, \ldots, b_n$, one obtains a propositional formula $\psi(b_1^1, \ldots, b_1^{m_1}, \ldots, b_n^1, \ldots b_n^{m_n}, a_1, \ldots, a_p)$ where b_i^j are the bits of the $b_1, \ldots, b_n$ bitvectors, and the a_i are auxiliary variables. The transformation satisfies two important constraints:

equisatisfiability ϕ is satisfiable if and only if ψ is.

deterministic auxiliary variables For all partial valuations m of the b_i^j, the formula induced by this valuation $\psi|_m$ (which has the a_i as free variables) has either 0 or 1 models.

This implies that any satisfying interpretation and assignment of ϕ corresponds to exactly one model of ψ even in the presence of auxiliary variables, and vice versa. This allows us to define the model count of a bitvector formula ϕ as the model count of the corresponding bitblasted formula ψ, even though in the general case of SMT this would be problematic.

Arrays The theory of arrays is one way of implementing a map from an address to a bitvector modeling the memory of the program. For each pair of sorts (s, t) the theory adds a sort $\text{Array}(s, t)$ representing a mapping from s elements to t elements. The theory adds two function symbols

- $\text{store} : \text{Array}(s, t) \times s \times t \to \text{Array}(s, t)$: $\text{store}(a, i, e)$ represents the array identical to a, but where the element at index i is replaced by e;
- $\text{select} : \text{Array}(s, t) \times s \to t$: $\text{select}(a, i)$ is the element stored at index i in a.

It also provides an equality predicate between arrays, but we do not focus on this as we only manipulate one array for memory. The main axiom of the theory of arrays is the read over write axiom:

$$\text{select}(\text{store}(a, i, e), j) = \text{ite}(i = j, e, \text{select}(a, i, e))$$

where $\text{ite}(c, a, b)$ denotes "**I**f c **T**hen a **E**lse b". Applying this axiom eagerly allows to remove the theory of arrays from any formula, even though this is usually not the most efficient way to solve it.

Adding the theory of arrays to another theory can make it undecidable, and there has been works to determine fragments of the theory of arrays which remain decidable [19]. In our specific case however, the domain of arrays of bitvectors is finite, and the theory is therefore decidable.

In the end, to determine the satisfiability of a formula in the theory of arrays and bitvectors, we can either use dedicated SMT solver like Z3 [48] or Boolector [97], or obtain a propositional formula by read-over-write and bitblasting. For model counting, there are only few solvers which can handle these theories [25], and experimentally bitblasting has proven more practical [18] despite the lack of sophistication of this approach.

Chapter 4

Robust reachability

In this chapter we present a first attempt at refining reachability to remove false positives in practice. It takes the form of *robust reachability*, a stronger property expressing that when the attacker chooses the optimal controlled input, he reaches his goal 100% of the time. We discuss this property both formally and in terms of proof methods, with a number of case studies.

4.1 Introduction

Context Many problems in software verification are encoded as *reachability* queries of some undesired condition—a bug, the exploitation of a vulnerability, *etc.* When a verification engine establishes that a certain buggy location in the program is reachable, an input triggering the bug is reported to the developer so that it can be fixed. In the case of techniques based on an under-approximation of program behaviors, like Symbolic Execution (SE) [22] or Bounded Model Checking (BMC) [29], we even have *in principle* the guarantee that the reported issue is real (*correctness*): there are no false positives.

Problem Yet, things are more subtle in practice, as some bugs can be triggered reliably whereas others only happen in very specific and highly improbable initial conditions. While standard reachability cannot tell the difference, this distinction is crucial in many real-life scenarios related to security (bug triage, bug prioritization, criticality assessment) or software engineering (test suite replicability and the problem of flaky tests [91]). For example, fuzzers are able to detect so many bugs [74] that they can lead to "bug triage issues" [64]. If each *replicable* (reliably-triggered) bug is hidden by dozens of more *fragile* ones in the reports of a verification engine, it is hard to focus development effort efficiently. Also, if one is only interested in vulnerability reports, bugs which cannot be reliably triggered may even be dismissed as "not exploitable" altogether.

Goal & challenges *Our goal is to develop a formal framework able to distinguish replicable bugs from fragile bugs, and amenable to automatic software verification — precisely, we want to be able in practice to find such replicable bugs.* This leads us to avoid any quantitative [73] or probabilistic reasoning [70, 5], insofar as their computational cost would hinder automation on real examples. We will revisit this trade-off in Chapter 5.

Proposal Our approach consists in partitioning inputs of the program into *controlled inputs* and *uncontrolled inputs*. This lets us refine the concept of reachability into *robust reachability*: a (buggy) location of a program is robustly reachable if there exist controlled inputs, such that for all uncontrolled inputs, this location is reached. In other words, with adequate input we do not need luck.

We typically focus on *security* scenarios where an *attacker* provides controlled input in one go, without knowledge of uncontrolled input — typically sending a malicious crafted file to obtain remote code execution or privilege escalation. We *deliberately* exclude interactive attack scenarios and weaker interpretations like "bugs replicable *most of the time*" in order to keep proof methods *tractable*.

Proving robust reachability is harder than standard reachability. While we show that robust reachability is expressible in formalisms like branching temporal logics [30], hyperproperties [32] or hyper temporal logic [33], there exist no efficient automated analysis methods for these formalisms at the software level (for Turing-complete languages). Therefore, we investigate dedicated verification techniques, revisiting standard methods (SE, BMC) for standard reachability as well as some of their standard companion optimizations.

Our prototype of Robust Symbolic Execution (RSE) relies on the ability of state of the art SMT solvers [9] to generate models for *universally* quantified formulas [58, 56, 104], which comes with a performance and completeness cost — yet we report promising results.

Contributions We claim the following contributions.

- We formally introduce the concept of robust reachability (Section 4.4) along with the more general robust safety and guarantee property, and motivate its use (Section 4.2), giving practical examples where standard reachability leads to false positives in practice (whatever the underlying verification technology). We also characterize robust reachability in terms of temporal logic and hyperproperties, and compare it with *non-interference* (Section 4.4);

- We revisit Symbolic Execution (SE) [22] and Bounded Model Checking (BMC) [29] and show how they can be lifted to the robust case (Section 4.5). While they both have the same deductive power in the standard case, they do not anymore in the robust setting — yet, *path merging* allows Robust SE to pace up with Robust BMC. Finally, we show how to adapt standard optimizations for Symbolic Execution and Bounded Model Checking;

- We implement and evaluate[1] (Section 4.6) the *first* symbolic execution engine dedicated to robust reachability, namely Binsec/RSE. We show how to use it for criticality assessment of 5 existing vulnerabilities (CVEs), and compare it with standard symbolic execution. RSE appears to be tractable with reasonable overhead, yielding false-positive-free symbolic reasoning.

We believe robust reachability is an important sweet spot in terms of expressiveness and tractability, allowing to highlight serious bugs in practical situations. We hope this first step will pave the way to more refinements and applications of robust reachability.

4.2 Motivation

In this section we show why standard reachability is not always a good fit for bug finding, as it cannot distinguish between *replicable* bugs and *fragile* bugs.

[1] The tool, benchmark and data are available at https://github.com/binsec/cav2021-artifacts and https://zenodo.org/record/4721753.

```
void fill(unsigned n, char* ptr) {
 for (unsigned i = 0; i < n; i++) {
   ptr[i] = 0x61;
 }
}
void victim() {
 unsigned n = controlled_input;
 char buffer[8];
 fill(n, buffer);
}
void main() {
 victim();
}
```

(a) C-like code, for simplicity

```
1  void victim() {
2   /* stack variables, top to bottom */
3   // return address goes here
4   int canary = global_random_value;
5   char buffer[8];
6   /* end stack variables */
7
8   register unsigned n = controlled_input;
9   fill(n, buffer);
10  if (canary != global_random_value)
11   fail_and_dont_return_at_all();
12  /* everything is ok */
13 }
```

(b) Explanation of compiler instrumentation with Stack Smashing Protection (SSP)

Figure 4.1: Simple stack buffer overflow

Stack canaries Consider the program presented in Figure 4.1. It suffers from a stack buffer overflow: if variable `n` is greater than 8 (the size of `buffer`), then `0x61` will be written to stack memory above `buffer`. For high enough `n`, this will overwrite the return address (Figure 4.1b, line 3) of function `victim` and make the program jump to an unexpected program location when `victim` returns.

Mitigations for such programming errors exist, like Stack Smashing Protection (SSP) [36]. This technique consists in pushing a randomly-chosen constant value called a *canary* at the top of the stack in the prologue of each function, and checking that this value is intact before returning. If the canary has been tampered with, the program exits to prevent exploitation (Figure 4.1b, line 11). Here, SSP prevents the attacker from overwriting the return address of `victim`, as doing so also overwrites the canary with `0x61616161`. This will be detected at line 10 of Figure 4.1b with probability $1 - 2^{-32}$ on a 32-bit architecture: the only way to pass through it is to have the canary value equal to `0x61616161`. *Hence, the buffer overflow in this program is not exploitable anymore.*

Table 4.1: Standard reachability is not a good criterion to measure the protection of SSP on the program of Figure 4.1.

Prog. Fig. 4.1	Ground truth	Standard reachability	Binsec [49]	Angr [114]	Robust reachability	Binsec/RSE
No SSP	vulnerable	vulnerable ✓	vulnerable ✓	vulnerable ✓	vulnerable ✓	vulnerable ✓
SSP	protected	vulnerable ✗	vulnerable ✗	vulnerable ✗	protected ✓	protected ✓

The problem with standard reachability Can the attacker hijack the control flow without triggering SSP? We can model this security question as a *standard reachability query* over inputs `controlled_input` and `global_random_value`. The attacker succeeds if line 12 is reachable with the additional condition that the return address of `victim` is overwritten with an unexpected address.

This standard reachability query is satisfiable with the canary `global_random_value` equal to `0x61616161` and `controlled_input` equal to *e.g.*, 42. And indeed, binary-level SE tools Angr [114] or Binsec [49] do report the bug as reachable (cf. Table 4.1). However, this answer is unsatisfying as this only happens with a very low probability: it may not be considered a plausible attack.

Hence, it turns out that SE can yield false positives in practice — especially in a security context.

Proposal: robust reachability We attempt to take the capabilities of the attacker into account as follows: We label `controlled_input` as a *controlled input* and `global_random_value` as an *uncontrolled input*. There exists no value of `controlled_input` such that `victim` returns to an address tampered with independently of the value of `global_random_value`. We thus say that our exploitation condition (line 12) is not *robustly reachable*. We can automatically verify this intuition. We adapted the SE engine of Binsec to robust reachability: our tool finds the vulnerability when we disable the protection (by labelling the canary as *controlled input*) and does not find it anymore when the protection is present. This shows that robust reachability can model the protection provided by SSP, while standard reachability cannot.

This phenomenon is not restricted to stack protectors. We identify in Table 4.2 several situations where standard reachability may lead to false positives, unlike robust reachability. Note that some cases (randomization based protections, uninitialized reads) concern binary-level issues, and cannot be observed from a source-level analysis.

Discussion Consider the slightly different problem of reaching line 11 in Figure 4.1b. It is reachable for all values of the canary *except* `0x61616161`, hence it is not considered robustly reachable — *all* values of uncontrolled input should lead to line 11. This restriction is *deliberate*. We will attempt to lift it in Chapter 5, where we show that it makes solver queries 7 times slower on average, with more timeouts. For similar reasons, we limit ourselves to non-interactive scenarios, where the attacker input is chosen before uncontrolled input are known. We will further motivate this choice in Sections 4.4.1 and 4.6.4.3.

Despite these deliberate restrictions, our case studies (Section 4.6.2) show the versatility of robust reachability. In the example above, we distinguish inputs controlled by an attacker (a bad guy) from inputs which he cannot influence (see also *e.g.* libvncserver in Section 4.6.2). But with `doas` (Section 4.6.2), we distinguish inputs controlled by the system administrator (the good guy) from those which vary on each execution. Other situations are possible, for instance deterministic inputs versus non-deterministic ones like in the case of flaky tests [91] — where there are neither good nor bad guys. Robust reachability can help in all these situations either assessing the "quality" of a given trigger or test suite (criticality, replicability), generating "good" triggers or test suites, or proving their absence.

4.3 Background

We recall some notations and definitions of Section 3.1: We consider a program P and $\mathcal{S}$ the set of its possible states. The initial state $s_1(y)$ is determined by the program input $y \in \mathcal{Y}$. The program P is represented as the set of the traces, *i.e.* sequence of states, that it can generate: $T(P) \subseteq \mathcal{S}^\infty$, where $\mathcal{S}^\infty$ denotes the set of all possible traces, finite or infinite. Finite traces are denoted as $\mathcal{S}^+$. We denote $t \prec t'$ the fact that trace t is a prefix of trace t'. Reachability is usually understood as control flow going through a location ℓ in the program, for example "line 13 in `file.c`". The location corresponding to a state $s \in \mathcal{S}$ is $\lambda(s) \in \mathcal{L}$. We use *trace* for successions of states and *path* for successions of locations. By abuse of notation, the path corresponding to a trace $t \in \mathcal{S}^+$ is $\lambda(t) \in \mathcal{L}^+$.

One often needs to consider more complex events than mere reachability of a location like reaching a location of the program with some additional condition on the past of the program, as is the case in our motivating example. For this reason we take a more general definition.

Table 4.2: Program constructs for which standard reachability yields fragile input

Randomization based protections	Standard reachability models randomized or arbitrary values like canaries or ASLR as attacker-chosen values. This voids such protections. See also Figure 4.1 and `libvncserver` in Section 4.6.2.
Uninitialized reads	With standard reachability, the attacker can choose the initial content of uninitialized memory. For example, he can choose it to contain a password or a secret. See also `doas` in Section 4.6.2.
Underspecified initial state	A bug which is unreachable in normal operating conditions can become reachable if, *e.g.*, one leaves the stack location completely free. Then the bug only happens with pathological initial state.
Undefined behavior	A bug in a branch depending on undefined behavior is still *technically* reachable, but not robustly reachable. Note that even machine code has some undefined behaviors.
Interactions with the environment	Contrary to robust reachability, standard reachability lets the attacker use system calls and interactions by *e.g.* letting him choose the date to nanosecond precision, as if the environment helped him.
Opaque functions	One can abstract complex functions (crypto functions, `malloc`) as black boxes returning a fresh, symbolic value. Standard reachability allows the attacker to choose these values, yielding fragile triggers.

Definition 1 (Reachability)**.** A set of finite traces $O \subseteq \mathcal{S}^+$ is reachable in program P if $P \cap O \neq \varnothing$. We write $P \vdash R(O)$.

As a special case one gets the usual property of reachability of a location:

Definition 2 (Reachability of a location)**.** A location is reachable, denoted as $P \vdash R(\ell)$, if $P \vdash R(O)$ with $O = \{t \in \mathcal{S}^+ \mid \lambda(t_{|t|}) = \ell\}$.

Finally, for a program $P \subseteq \mathcal{S}^\infty$, the restriction $P|_y$ of this program to input y is defined by $T\left(P|_y\right) \triangleq \{t \in T(P) \mid t_1 = s_1(y)\}$, the restriction $P|^\pi$ of P to path π by $T(P|^\pi) \triangleq \{t \in T(P) \mid \exists t' \in \mathcal{S}^+.\, t' \prec t \wedge \lambda(t) = \pi\}$, and the restriction $P|^{\leq k}$ of P to bound $k \in \mathbb{N}$ by $T\left(P|^{\leq k}\right) \triangleq \{t \in T(P) \mid |t| \leq k\}$.

Symbolic Execution (SE) and Bounded Model checking (BMC) Consider the problem of proving or disproving reachability of $O \subseteq \mathcal{S}^+$. In general, this problem is undecidable, so verifiers cannot be both correct and complete (see Section 3.1.3). Correct verifiers can still be k-complete as k-completeness can be thought of as completeness for finite-path systems. Let us describe two such methods, which consist in encoding the possible witnesses in O as a SMT formula.

SE [22] incrementally explores all paths in the program (up to, say, a bound k). Each path π is converted into a SMT formula pc_π^O, called *path constraint*. It has input y as sole free variable and expresses that when executed with input y, the program does indeed follow the path π and starts with a prefix in O. SE looks for a witness in O by iteratively checking satisfiability of pc_π^O for all enumerated π. Conversely, BMC [29] considers the program as a whole (unrolled up to a bound k) and builds a SMT formula expressing that it contains a trace prefix in O. This formula is actually equivalent to the disjunction of the path constraints of these paths. These algorithms

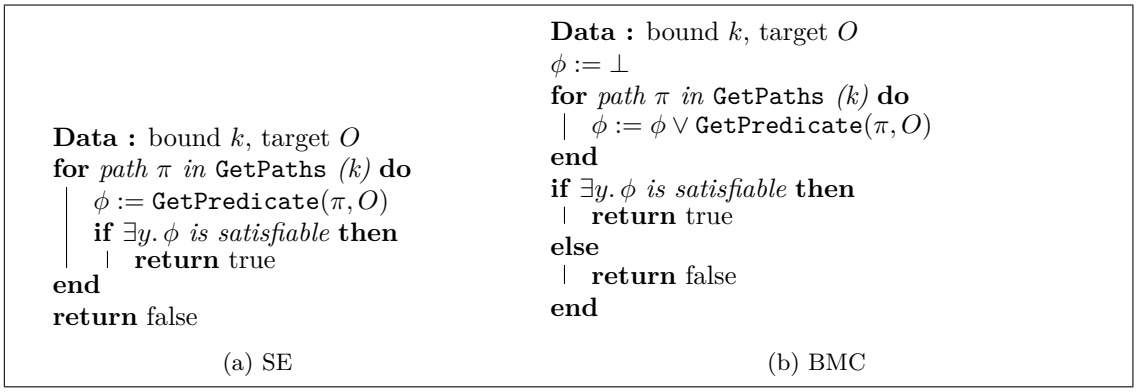


Figure 4.2: Bounded proof attempt of $R(O)$ with SE and BMC

are given in Figure 4.2, where $\texttt{GetPredicate}(\pi, O)$ turns a path into its path constraint pc_π^O and $\texttt{GetPaths}(k)$ yields all paths below size bound k.

Proposition 1. *SE and BMC have the same expressive power: both are correct and k-complete with respect to reachability properties.*

Interestingly, we show in Section 4.5 that this is not true anymore with robust reachability.

Solvers SE and BMC commonly discharge their satisfiability queries to SMT solvers [9] which take formulas as input, and output whether they are satisfiable (along with a model) or not. Typical queries are expressed in the quantifier-free fragments of well known theories (linear integer arithmetic, bitvectors, arrays, *etc.*) where SMT solvers perform well in practice. In case of an undecidable theory, we can use incomplete solvers (possibly answering UNKNOWN), at the price of k-completeness.

4.4 Robust reachability

In this section we provide a formal definition of robust reachability, and argue why it deserves being singled out as a new problem rather than being viewed as a special case of a more generic framework like some of the more expressive temporal logics.

4.4.1 Definition

We introduce the new notion of *robust reachability*. We partition the input y into the *controlled input* a and the *uncontrolled input* x — we denote $y \triangleq (a, x)$. Let $\mathcal{A}$ and $\mathcal{X}$ be the sets of possible controlled and uncontrolled inputs respectively. A location is *robustly reachable* when the attacker can choose controlled input $a \in \mathcal{A}$ without having to rely on specific values of the uncontrolled input $x \in \mathcal{X}$ to reach his target. Input a is then called a *robust trigger* — otherwise it is a *fragile trigger*.

Definition 14 (Robust reachability)**.** A set of finite traces $O \in \mathcal{S}^+$ is robustly reachable in program P, denoted as $P \vdash \mathfrak{R}(O)$, if

$$\exists a \in \mathcal{A}. \forall x \in \mathcal{X}.\ P|_{(a,x)} \vdash R(O)$$

Proposition 2. *Robust reachability implies standard reachability. The converse implication does not hold.*

Discussion As already mentioned at the end of Section 4.2, our definition of robust reachability specifically targets a threat model where the attacker speaks first, unaware of uncontrolled inputs. It deliberately excludes interactive systems where the attacker can choose some input, then receive some program output possibly leaking uncontrolled input, and then choose some more input *depending on what was received*. Modeling such situations requires additional quantifier alternations, which deeply impact the performance of proof methods and cripple automation, as shown in Section 4.6.4.3.

Likewise, a bug triggered for all uncontrolled inputs but one is not robustly reachable according to Definition 14. A quantitative definition of robust reachability could take into account the *proportion of uncontrolled inputs* triggering a bug. This will be developed in Chapter 5 as an alternative that trades some performance for additional precision.

In other words, Definition 14 is a trade-off to keep robust reachability amenable to automated verification. This does not prevent it from meeting its main goal: drawing the attention on more serious bugs. Some may of course be missed, but, as our case studies will show (Section 4.6), a good number will be found.

In the rest of this section, we review a few related properties and see how much they overlap with, but do not remove the need of, robust reachability.

4.4.2 Relation with non-interference

We partition inputs and outputs of a system into either *high* (highly classified) or *low* (public, *e.g.* observable). A system satisfies *non-interference* [65] when low outputs do not depend on high inputs, implying that secrets cannot leak. Robust reachability can be reformulated in a very non-interference-sounding phrasing: uncontrolled inputs (call them high) must not interfere with the attacker reaching his goal (the low output). Let us clarify this link, focusing on robust reachability of a location ℓ for simplicity.

Formally, let high input be uncontrolled input x, and low input be controlled input a. Let low output be whether control flow reached location ℓ. Non-interference of the resulting system means that

$$\forall a, x, x'.\ \Big(P|_{(a,x)} \vdash R(\ell) \iff P|_{(a,x')} \vdash R(\ell) \Big)$$

Proposition 3. *If ℓ is (standardly) reachable and the system satisfies non-interference with the high/low partition described above, then ℓ is robustly reachable. The converse is false.*

Robust reachability requires a single value of the controlled input a for which reachability of ℓ is guaranteed but says nothing for other values of a, where observable behavior may depend on uncontrolled input x. On the other hand, non-interference constrains the system to behave much more independently of uncontrolled input, even for "uninteresting" values of a. Additionally, non-interference says nothing of reachability.

4.4.3 Interpretation in terms of hyperproperty

Robust reachability and its negation are not trace properties: the observation of a single trace is never enough to prove or disprove them. For example, observing a single trace reaching target ℓ with input (a, x) is both compatible with ℓ being robustly reachable (if all other inputs $(a, x'), x' \in \mathcal{X}$ also reach ℓ), and with ℓ not being robustly reachable (if some other x' is such

that (a, x') does not reach ℓ). In such case one often resorts to the formalism of hyperproperties introduced by Clarkson and Schneider [32]. A hyperproperty Π is represented as the set of programs P that satisfy it in the form of their set of traces: $\{T(P) \mid P \vdash \Pi\}$. Clarkson and Schneider [32] also show that any hyperproperty is the intersection of a hypersafety hyperproperty (*i.e.* something bad cannot happen) and a hyperliveness hyperproperty (something good will eventually happen). Hypersafety is generally thought as easier to prove, notably with self-composition [12]. Unfortunately, robust reachability and its negation are pure hyperliveness in the general case: no finite set of finite traces can falsify them. However, in some conditions, they degenerate partly into hypersafety:

Proposition 4. *If the domain $\mathcal{X}$ of uncontrolled inputs is finite, then the negation of robust reachability is not pure hyperliveness (*i.e.*, it has a non-trivial hypersafety component).*

Proof. Robust reachability of $O \subseteq \mathcal{S}^+$ can be proved by finding controlled input $a \in \mathcal{A}$ such that for all uncontrolled input $x \in \mathcal{X}$ one observes a trace starting with input (a, x) and belonging to O. When $\mathcal{X}$ is finite, this means that a finite observation can disprove non-(robust reachability). This is the definition of hypersafety. □

This idea—trying to observe a hopefully small set of traces which together prove robust reachability—is crucial for algorithms and leads to our use of path merging in Section 4.5.3.

4.4.4 Interpretation in terms of temporal logic

We now show how robust reachability can be expressed by some sufficiently expressive temporal logics. We consider robust reachability of a location ℓ instead of an arbitrary set of traces O in order to focus the discussion on robustness itself.

Computational Tree Logic (CTL) CTL [30] is a temporal logic over the tree of possible traces. Let L be a labeling which maps states to the set of (atomic) predicates they satisfy. If ℓ is a predicate, the CTL formula ℓ is satisfied by all systems whose initial state s_0 verifies $\ell \in L(s_0)$. If ϕ is a CTL formula and s a state, then $\mathbf{EX}\phi$ expresses that ϕ holds in at least one (direct) successor of s, and $\mathbf{AF}\phi$ that all traces arising from s eventually reach a state from which ϕ holds. CTL introduces other operators, not needed here.

Proposition 5. *CTL can express robust reachability of location ℓ.*

Proof. Let $\mathcal{S}' \triangleq \mathcal{S} \cup \mathcal{A} \cup \{s_i\}$ where s_i is a new state, let $\rightarrow' \triangleq \rightarrow \cup \{(s_i, a) \mid a \in \mathcal{A}\} \cup \{(a, s_1(a, x)) \mid a \in \mathcal{A}, x \in \mathcal{X}\}$, and let $L'(s)$ be equal to $L(s)$ if $s \in \mathcal{S}$ and $\varnothing$ otherwise. Then ℓ is robustly reachable if, and only if $\mathbf{EXAF}\ell$ is true in the new extended system $(\mathcal{S}', \rightarrow', L')$ with s_i as initial state. □

Alternating-Time Temporal Logic (ATL) ATL [1] is a temporal logic designed to model systems with multiple actors with distinct objectives. As usual the system is modeled by its set of states and its transition function, but each transition is decided by a set Σ of players: each player simultaneously makes a decision, and the actual transition is selected depending on these decisions. ATL formulas are the same as CTL, but operators $\mathbf{A}$ and $\mathbf{E}$ are generalized by a new operator $\langle\langle\cdot\rangle\rangle$. For a set of player $\Lambda \subseteq \Sigma$, $\langle\langle\Lambda\rangle\rangle\varphi$ means that there exists a strategy for players in Λ to make the system satisfy φ. At the limit, $\langle\langle\varnothing\rangle\rangle\varphi$ is $\mathbf{A}\varphi$ and $\langle\langle\Sigma\rangle\rangle$ means $\mathbf{E}\varphi$.

ATL contains CTL so Proposition 5 applies. However, ATL makes it much more natural to express robust reachability since players can oppose each other. Consider $\Sigma = \{\epsilon, \alpha\}$ where ϵ is the environment, and α the attacker. Reachability of ℓ is $\mathbf{EF}\ell$, or written otherwise: $\langle\langle\Sigma\rangle\rangle\mathbf{F}\ell$,

which means that both the environment and attacker can cooperate to reach ℓ. Robust reachability on the other hand is $\langle\!\langle\{\alpha\}\rangle\!\rangle \mathbf{F}\ell$, meaning that the attacker alone can make it so ℓ is reached.

Similarly, one can express the negation of robust reachability: $\langle\!\langle\{\epsilon\}\rangle\!\rangle \mathbf{G}\neg\ell$, which is a particular case of "collaborative invariance" [1].

HyperLTL It is also possible to express robust reachability in the temporal logic HyperLTL[33], which allows to reason over sets of traces π, assuming we have an atomic predicate $\equiv_v$ stating that the first states of two traces have the same value for variable v. Robust reachability of ℓ can then be expressed as $\exists\pi.\,\forall\pi'.\,\mathbf{F}\ell_\pi \wedge (\pi \equiv_a \pi' \rightarrow \mathbf{F}\ell_{\pi'})$, where $\mathbf{F}\ell_\pi$ denotes that trace π goes through ℓ. In other words, there exists a trace π reaching ℓ such that all traces sharing the same controlled input also reach ℓ.

4.4.5 Robust reachability and automatic verification

The previous classification does not help us find an efficient *software verification method* for robust reachability. Indeed, while efficient CTL model checkers exists for the finite case [28] or very specific formalisms such as push-down systems [116], most efforts in (general) software verification have been directed towards the verification of safety temporal formulas or simple termination [34] (formulas of the form $\mathbf{AF}\varphi$). HyperLTL [33] suffers the same limitations. As for ATL, it is so expressive (there can be arbitrarily many players, and arbitrarily many interactions between them and the system) that state-of-the-art tools like STV [79] are limited to small models of a few dozens of states at best.

Moreover, checking for both reachability and non-interference as a correct, but incomplete proof method for robust reachability is probably too incomplete in practice. Finally, one can prove the *absence* of robust reachability by proving the absence of standard reachability. It is thus possible to use existing algorithms for unreachability, based *e.g.* on invariant computation, at the price of even larger over-approximation than when they are used for their original purpose. This kind of approach is not our focus. In this chapter we look for *correct verifiers* able to prove robust reachability (and report robust triggers) rather than to disprove it.

4.5 Automatically proving robust reachability

We now discuss how to extend SE and BMC to the robust case.

4.5.1 Robust Bounded Model Checking

As mentioned in Section 4.3, BMC determines the reachability of $O \subseteq \mathcal{S}^+$ by building a family of SMT formulas $\varphi_k(a,x)$ equivalent to $P|^{\leq k} \vdash R(O)$. In the case of reachability of a location ℓ, φ_k expresses that ℓ is reachable in less that k steps. Then one proves that $R(O)$ holds if and only if $\exists k.\,\exists a.\,\exists x.\,\varphi_k(a,x)$. This extends to robust reachability:

Proposition 6. *If the domain of uncontrolled input $\mathcal{X}$ is finite or P has finitely many paths, then $P \vdash \mathfrak{R}(O)$ if and only if $\exists k.\,\exists a.\,\forall x.\,\varphi_k(a,x)$.*

Proof. ($\Longleftarrow$) comes directly from the definition of φ_k. ($\Longrightarrow$). If ℓ is robustly reachable, let a_0 be a robust trigger. The set of paths W arising from inputs in $\{a_0\}\times\mathcal{X}$ is finite (bounded either by $\mathcal{X}$ or the number of paths in the system), and $\forall x.\ \bigvee_{\pi\in W} \mathrm{pc}^O_\pi(a_0,x)$ holds. Let $k = 1+\max_{\pi\in W}|\pi|$. All paths in P are unrolled in φ_k so $\bigvee_{\pi\in W} \mathrm{pc}^O_\pi(a_0,x) \implies \varphi_k(a_0,x)$ and thus $\forall x.\,\varphi_k(a_0,x)$. □

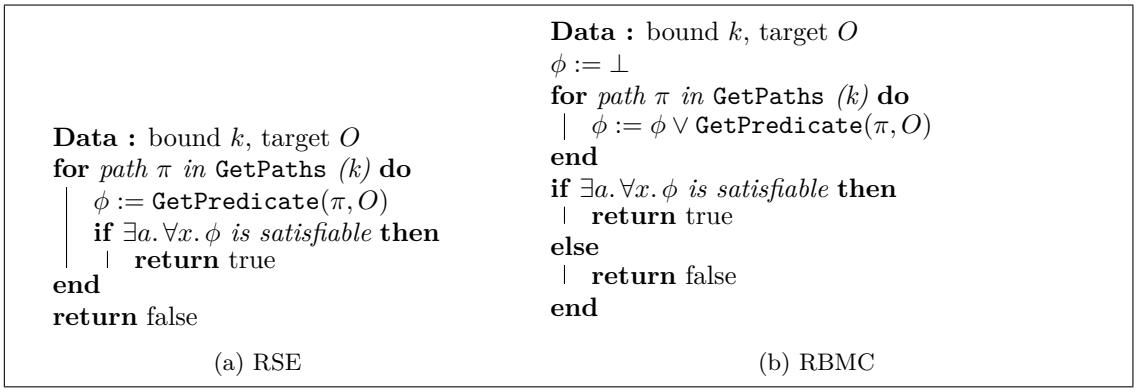


Figure 4.3: Lifting SE and BMC to robust reachability

As a result, it is enough to replace the condition "$\exists y.\, \phi$ is satisfiable" by "$\exists a. \forall x.\, \phi$ is satisfiable" in Figure 4.2b. The resulting algorithm, called Robust BMC is presented in Figure 4.3b.

Corollary 1. *Robust BMC is correct w.r.t. robust reachability. If the domain of uncontrolled input $\mathcal{X}$ is finite or the system has finitely many paths, then robust BMC is also k-complete.*

The finiteness hypothesis is required: if a program reaches a location after having executed a loop an unbounded, uncontrolled number of times, then robust BMC has to unroll an unbounded number of paths to prove robust reachability.

4.5.2 Robust Symbolic Execution

Similarly to BMC, we check that a path π robustly reaches the target by checking the satisfiability of $\exists a. \forall x.\ \mathrm{pc}_\pi^O(a, x)$, instead of $\exists a. \exists x.\ \mathrm{pc}_\pi^O(a, x)$. This means replacing "$\exists y.\, \phi$ is satisfiable" by "$\exists a. \forall x.\, \phi$ is satisfiable" in Figure 4.2a. Unfortunately the resulting algorithm, robust SE (Figure 4.3a), is not exactly what we want, as it proves a stronger property.

Definition 15 (Single-path robust reachability)**.** A set O is single-path robustly reachable if $\exists \pi \in \mathcal{L}^+. \exists a. \forall x.\ P|^\pi|_{(a,x)} \vdash R(O)$. In other words, the path used to reach O is the same regardless of the uncontrolled input.

Proposition 7. *Single-path robust reachability implies robust reachability. The converse implication does not hold.*

Proposition 8. *Robust SE is correct and k-complete w.r.t. single-path robust reachability.*

Proof. By construction, $\mathrm{pc}_\pi^O(a, x)$ is equivalent to $P|^\pi|_{(a,x)} \vdash R(\ell)$. $\exists \pi. \exists a. \forall x.\ \mathrm{pc}_\pi^O(a, x)$ is therefore equivalent to single-path robust reachability of the last location of π. □

Corollary 2. *Robust SE is correct but incomplete for robust reachability.*

Interestingly, the expressive powers of SE and BMC, which are the same for standard reachability, diverge when extended to robust reachability.

4.5.3 Path merging

Path merging [69] (a.k.a. state joining) consists in identifying "close" paths leading to the same location and replacing them by a *merged* path (summary). With original path constraints $\mathrm{pc}^O_{\pi_1}$ and $\mathrm{pc}^O_{\pi_2}$, the merged path constraint is $\mathrm{pc}^O_{\pi_1} \vee \mathrm{pc}^O_{\pi_2}$. This is only an optimization in the standard setting, with no impact on k-completeness. The situation is different in the robust setting.

Data : bound k, target O
1 $\phi := \bot$
2 **for** *path* π *in* `GetPaths` *(k)* **do**
3 $\quad \phi := \phi \vee$ `GetPredicate`(π, O)
4 $\quad$ **if** $\exists a. \forall x. \phi$ *is satisfiable* **then**
5 $\quad\quad$ **return** true
6 **end**
7 **return** false

Algorithm 1 : RSE+: Robust SE with systematic path merging

```
1  void main(a, x) {
2    if (x) x++; // π1
3    else x--;   // π2
4
5    if (!a) bug();
6  }
```

Figure 4.4: An example where path merging is required

Consider the program in Figure 4.4: the bug is robustly reachable with controlled input $a = 0$, but the control flow takes one of two paths π_1 and π_2 depending on the value x of uncontrolled input. This bug will not be found by robust SE as defined previously, as neither π_1 nor π_2 fulfills the satisfiability criterion $\exists a. \forall x.\ \mathrm{pc}^O_{\pi_i}(a, x)$. However, if π_1 and π_2 are merged, then the bug is found because $\exists a. \forall x.\ \mathrm{pc}^O_{\pi_1}(a, x) \vee \mathrm{pc}^O_{\pi_2}(a, x)$ is satisfiable. This leads us to robust SE with systematic path merging (RSE+, Algorithm 1), better fit to robust reachability.

Proposition 9. *Robust SE with systematic path merging (RSE+) is correct for all robust reachability properties. If the domain of uncontrolled input $\mathcal{X}$ is finite or the system has finitely many paths, then it is also k-complete.*

Proof. For k-completeness: If $\mathfrak{R}(O)$ holds, let a_0 be a robust trigger. The set of paths P arising from inputs in $\{a_0\} \times \mathcal{X}$ is finite (bounded either by $\mathcal{X}$ or the number of paths in the system). Let $k = 1 + \max_{\pi \in P} |\pi|$. For bound k, when `GetPaths` has output all paths in P, $\bigvee_{\pi \in P} \mathrm{pc}^O_\pi \implies \phi$ so $\exists a. \forall x. \phi$ is satisfiable. □

In conclusion, *path merging improves the completeness of robust SE*. This is surprising because path merging is merely optional in standard SE.

4.5.4 Revisiting standard optimizations and constructs

Some optimizations commonly used in SE are not correct nor complete anymore in a robust setting. We show here how to adapt them.

Incremental path pruning [121, 8] $\mathcal{S}^+$ is the always reached objective. Therefore, $\mathrm{pc}^{\mathcal{S}^+}_\pi$ expresses that a path π is executable. We can use this to perform an optimization called *incremental path pruning*. When a path has an unsatisfiable path constraint $\mathrm{pc}^{\mathcal{S}^+}_\pi$, all its descendant paths are also infeasible. For example, the path `acd` in Figure 4.5 has path constraint $x < 10 \wedge x > 20$, which is unsatisfiable. One can prune this path, *i.e.* stop exploring it and its children `acde` and `acdf`.

```
Data : program entry point ℓ0, bound k
1 P := {ℓ0}
2 while P ≠ ∅ do
3 |   Take a path π out of P
  |   /* If too long, discard π          */
4 |   if |π| > k then continue
  |   /* pc_π^{S+} expresses that the path is
  |      executable                       */
5 |   if ∃a, x. pc_π^{S+} unsat then continue
6 |   yield π          // return π to caller
7 |   P := P ∪ {children paths of π}
8 end
```

Algorithm 2 : Implementation of `GetPaths` with path pruning

```
uncontrolled int x;
if (x<10)     { /* a */ }
else          { /* b */ }
/* c */
if (x>20) {
  /* d */
  if (x>30)   { /* e */ }
  else        { /* f */ }
}
```

Figure 4.5: Failure case for universal path pruning

```
Data : entrypoint ℓ0, bound k
P := {ℓ0}
while P ≠ ∅ do
|   Take a path π out of P
|   if |π| > k then continue
|   if ∃a.∀x. pc_π^{S+} unsat then
|   |   /* Skip MaybeMerge to
|   |      disable path merging  */
|   |   P := MaybeMerge(π, P)
|   |   continue
|   end
|   yield π
|   P := P ∪ {children paths of π}
end
```

Algorithm 3 : `GetPaths` with universal path pruning

```
1 Function MaybeMerge(π, P)
2 |   Choose u a transitive child of the last
  |     location of π (ideally, a strict
  |     postdominator of the second to last
  |     location of π)
3 |   Let π′ the longest strict prefix of π.
4 |   Let U the set of paths from π′ to u
5 |   if ∃a.∀x. ⋁_{π″∈U} π″ is SAT then
6 |   |   Merge paths in U and add the
  |   |     result to P
7 |   end
8 |   return P
```

Algorithm 4 : Incomplete path merging for universal path pruning

In Figure 4.2a this would be an optimization of `GetPaths`: as shown in Algorithm 2, one checks that the path constraint of currently explored paths are satisfiable, and if not, the paths at fault are *pruned*, and their children paths are not explored. As a result, we now issue satisfiability queries in two occasions: during `GetPaths` to *prune* paths (Algorithm 2, line 5), and when *validating* a candidate reaching path (Figure 4.2a, line 4). Pruning queries and validation queries must be treated differently.

Robust SE is obtained from SE by adding a universal quantifier to *validation queries* but not *pruning queries*. The path constraint for path `a` in Figure 4.5 is $\text{pc}_{\texttt{a}}^{\mathcal{S}^+} = x < 10$ but $\exists a. \forall x.\ \text{pc}_{\texttt{a}}^{\mathcal{S}^+}$ is false. Same applies for `b`. If we added a universal quantifier to pruning queries—which we call *universal path pruning*, see Algorithm 3—we would prune `a` and `b`, and incorrectly conclude that `c` is not robustly reachable. In other words, Symbolic Execution with universal path pruning (denoted $\text{RSE}_\forall$) is correct but not complete.

Universal path pruning, however, conveys an interesting intuition: the full `if` branch below `acd` in Figure 4.5 is not robustly reachable, because $\forall x. x > 20$ is false. With normal path pruning and RSE+, we would needlessly explore these paths. To take advantage of this, we keep $\text{RSE}_\forall$ but improve its completeness with path merging, as depicted in Algorithm 4.

The main idea is that when a set of paths are to be pruned, they may pass the universal pruning test $\exists a. \forall x.\ \text{pc}^{\mathcal{S}^+}$ when merged together. One way to find such sets of paths is to use the Control Flow Graph (CFG) of the program. For example when trying to prune $\pi = \texttt{a}$ in Figure 4.5, we know by invariant of the set P of paths to be explored that the empty path $\pi' = \epsilon$ passes the universal test. We compute the strict postdominator $u = \texttt{c}$ of π': when the paths from π' to c join again, they pass the pruning test again. We then replace π by this merged path in the set P of paths to be explored.

Note that computing a postdominator is not required for correction. In our implementation, we cannot compute the exact CFG at the binary level so the chosen u may be wrong. In line 5 of Algorithm 4 we check that we picked correctly, and otherwise, merging failed, and we prune π. Despite the heuristic approach, the technique proves useful, as we will see in Section 4.6.

We denote Robust SE with universal path pruning and path merging as $\text{RSE}_\forall$+. It is correct and "less incomplete" than $\text{RSE}_\forall$.

Assumptions It is common to model complex parts of the system by introducing their result as a symbolic input z and then *assume* that z satisfies the required properties. For example, ASLR for the stack pointer could be modeled by adding an *assumption* that $esp \in [m, M]$ where m and M are in-lined constant values. In standard SE this would be translated to an *assertion* $esp_0 \in [m, M]$ conjoined to the path constraint pc, where esp_0 is the initial value of esp. *Actually, in standard SE and BMC, assertions and assumptions are dealt with identically.*

```
controlled unsigned int a;
uncontrolled unsigned int x;
assume(x < a);
if (false) bug();
```

Figure 4.6: Unsound assumption, in pseudo-C.

In a robust setting, to the contrary, adding an *assumption* ψ to a path constraint yields $\psi \implies \text{pc}$, while adding an *assertion* ϕ yields $\text{pc} \wedge \phi$. Additionally, assumptions which mix controlled and uncontrolled inputs can make the algorithms above unsound without adaptation: in Figure 4.6, reachability of `bug` maps to the SMT query $\exists a. \forall x. x < a \implies \bot$. It is satisfiable, with $a = 0$, which makes the premise false. However, this does not correspond to an executable path. Actually, formalizing robust reachability assuming $\psi(a, x)$ naively by $\exists a. \forall x. (\psi(a, x) \implies P|_{(a,x)} \vdash R(\ell))$ does not imply standard reachability anymore. A slight adaptation is needed:

Definition 16 (Robust reachability under assumption)**.** O is robustly reachable *under the assumption of* ψ if

$$\exists a.\,((\exists x.\,\psi(a,x)) \wedge (\forall x.\,(\psi(a,x) \implies P|_{a,x} \vdash R(O))))$$

This definition preserves the implication from robust to standard reachability. The algorithms we presented are easily adapted to take it into account.

Interestingly, in the robust case, SE and BMC cannot handle assertions and assumptions in the same way anymore.

Concretization and other optimizations When path constraints along a path become too complex, some variables can be *concretized*: their symbolic value can be replaced by a concrete one [63, 109, 45]. Formally, concretizing a variable u to value 42 corresponds to adding an *assertion* $u = 42$. This sacrifices k-completeness for tractability. Actually, any additional constraint can be added, and several common optimizations (e.g., domain shrinking, path filtering) can be seen through this lens. These optimizations must be taken with care in the robust setting. First, considering them as assumptions instead of assertions would be incorrect. Second, if the value of the concretized variable ultimately depends semantically on uncontrolled input, the path does not pass universal validation anymore: for example, when concretizing x to 42, $\exists a.\,\forall x.\ \mathrm{pc}(a,x) \wedge x = 42$ is unsatisfiable because $\forall x.\,x = 42$ is false. As a result, locations visited further on this path become robustly unreachable. *In other words, concretization only works on controlled or constant values.*

4.5.5 About constraint solving

Adaptations to robust reachability require solvers to deal with one alternation of quantifiers. Most theories become undecidable with quantifiers. Dedicated algorithms exist for a few decidable quantified theories, *e.g.* the array property fragment [19] or Presburger arithmetic [20]. For other theories, generic methods like E-matching [47] and MBQI [58] have proven rather efficient, although not complete. Sound approximations [56] also have been proposed to reduce quantified formulas to quantifier-free ones. In our experiments, the newly introduced quantifier associates to an increase in the frequency of time-outs and memory-outs, as seen in Section 4.6.3 and specifically Table 4.4.

4.6 Proof-of-concept of a robust symbolic execution engine

4.6.1 Implementation

We propose Binsec/RSE, the *first* symbolic execution engine dedicated to robust reachability. We base our proof-of-concept on Binsec [49], a binary executable formal analysis engine written in OCaml and already used in several significant case studies [44, 40, 103]. For the sake of experimental evaluation (section 4.6.3) we actually implement five variants of robust reachability: **RSE** (basic approach in section 4.5.2 with existential path pruning Section 4.5.4), **RSE+** (the same plus systematic path merging, Section 4.5.3), **RSE**$_\forall$ (RSE with universal path pruning, Algorithm 3), **RSE**$_\forall$**+** (same, with path merging during path pruning, Algorithm 4), and **RBMC** (Section 4.5.1).

The source code of Binsec/RSE, the test suite and the case studies of this section are available for reproduction at https://github.com/binsec/cav2021-artifacts and https://zenodo.org/record/4721753.

Solving universally quantified SMT formulas BINSEC/RSE emits quantified formulas in the theory of bitvectors and arrays (arrays are used to model memory) which are then solved by the solver Z3 [48]. Z3 supports universally quantified formulas quite well, but has trouble handling cases where arrays are quantified. This is a problem as initial memory is an array, and in most threat models, should be labeled as uncontrolled. As an example, Z3 is actually not able to prove the unsatisfiability of a formula as simple as

$$\exists a.\, \forall mem.\, mem[42] = a \tag{4.1}$$

First we reuse the recent ROW simplification [55] to reduces the number of array indexations. In favorable cases, this simplification alone can even simplify all arrays out.

To deal with cases like (4.1) that remain after ROW simplification, we implemented one further simplification that moves the memory out of the universal quantifier. As the ROW simplification is quite powerful, it is only needed infrequently, and when it does, it is hard to reason about the root causes of the failure due to the complexity of the resulting formulas, but for the sake of illustration, let us modify one simpler test case to exhibit this behavior. Consider our function inversion test relating to musl's implementation of `strtol`. It looks for a controlled string s such that $\texttt{strtol}(s) = 42$. If we do not initialize the memory of an internal lookup table, then $\text{RSE}_\forall$ fails on a formula corresponding to

$$\exists s.\, \forall mem.\, mem[s[3] + table] \geq 10$$

where *table* is the offset of a table in the executable. Z3 returns UNKNOWN on this formula.

Let us now explain the transformation informally. In the case of eq. (4.1), we would like to rewrite it into $\exists a, mem.\, \forall v.\, (\text{store}(mem, 42, v))[42] = a$. This corresponds to initializing all memory locations that are later read with an uncontrolled, fresh value. As we are sure the original memory *mem* is never read, we can move it out of the universal quantifier. In our experience, Z3 deals easily with formulas where only bitvectors, as opposed to arrays, are universally quantified. The general case can be more involved, as the locations where memory is read can be symbolic, and even depend on memory. We therefore introduce one more layer of indirection:

$$\exists a.\, \forall mem.\, mem[mem[12] + 1] = a$$

would be transformed to

$$\exists a, mem.\, \forall v, i.\, \text{let } mem' = \text{store}(mem, i, v) \text{ in } i = mem'[12] + 1 \implies mem'[42] = a$$

When dealing with n reads inside the original formula, we need to introduce n symbolic indices, and deal with all possible equalities between them, so the transformation actually yields a formula of size $O(n^2)$. This can be problematic, but as already said, this is mostly a fallback when ROW does not simplify arrays out already. We thus expect very few memory reads to remain.

4.6.2 Case studies

4.6.2.1 Exploitability assessment for vulnerabilities

We show here how BINSEC/RSE (unless otherwise specified, the RSE+ variant) can help in vulnerability assessment. Especially, we demonstrate that robust reachability allows deeper insights into a bug than standard reachability, by replaying 5 existing vulnerabilities.

```
static int parseuid(const char *s, uid_t *uid) {
    const char *errstr;
    sscanf(s, "%d", uid);
    if (errstr) return -1;
    return 0;
}
```

This code is used to parse user IDs allowed to execute commands. If this function erroneously returns the attacker's user ID in the parameter `uid`, then privileged escalation is possible. When `s` is not the text representation of an integer, `uid` remains uninitialized memory. The branch `if (errstr)` was optimized out when we compiled. The exact same flaw is present in the function parsing group IDs.

Figure 4.7: Code responsible for CVE-2019-15900.

CVE-2019-15900 in doas `doas` is a utility granting higher privileges to users specified in a configuration file. User IDs are sometimes parsed incorrectly and left uninitialized. We look for a *vulnerable configuration file* denying root access to the attacker such that the (flawed) executable reliably grants root access to the attacker. For simplicity, we assume that the system has no named user and group, the configuration file has two lines and the attacker has uid 4 and gid 7.

BINSEC/RSE with standard reachability reports that root access is granted memory address `0xffefffff` contains the group ID of the attacker and the stack starts at `0xfff0001f`. *This is a typical "false positive in practice": these conditions may vary unpredictably across executions, so we cannot conclude regarding the exploitability of the flaw.*

With robust reachability where the configuration file is controlled but the initial state of memory is not, BINSEC/RSE reports in less than 10s that root access is granted reliably to the attacker when the configuration file contains `deny :4` and `permit b%@)@@(`. When parsing the first rule, `parseuid` correctly initializes the `uid` variable of Figure 4.7 to the uid of the attacker, 4.

The next rule allows root access to any non-existing username, `parseuid` leaves this variable untouched and privileged escalation is possible.

This result is considerably more useful, but `b%@)@@(` is not a valid username. We test therefore if any other given username is also affected by running the analysis with this username concretized in the initial state. By this method, we proved that the flaw is also robustly reachable for `wwww`, a possible typo of a usual username, as well as all two-letter lowercase usernames.

In other words, if the system administrator grants privileges to a non-existing user by mistake, he may unknowingly grant them to the attacker instead.

Additionally, BINSEC/RSE shows in about 4 minutes that the bug is not robustly reachable with only one line of configuration file. This proves that the model found by standard SE is not robust and that the model and attack trace found by RSE are significantly different. It is not the case that RSE found the same trace as SE and proved that it did in fact satisfy robust reachability; RSE had to search further for a qualitatively better report.

Here, robust reachability provides us with invaluable insight about the severity of a bug where standard reachability fails.

CVE-2019-20839 in libvncserver An attacker-chosen null-terminated string is copied by an unbounded `strcpy` into a 108-bytes buffer, leading to a stack buffer overflow. Exploitability is not guaranteed: null bytes cannot be copied, the executable is protected by SSP, *etc.* Starting from the vulnerable function, we ask whether it is possible to return to the address `0xdeadbeef`, chosen arbitrarily.

BINSEC reports that for standard reachability, the bug can be reached when: *(1)* the stack

starts at `0xfff00000`; *(2)* the initial value of the return address of the function is 0; *(3)* the `gs` segment starts at `0xf7f00000`; *(4)* the stack canary is `0x01010180`; *(5)* neither system call in the function fails; *(6)* file descriptor 0 is free; *(7)* the input path has a specific value. *The attacker cannot prepare such a state, so this is another false positive in practice.*

With robust reachability, when only the input buffer is controlled and not the stack canary, BINSEC/RSE fails to prove or disprove robust reachability in 24h. However, if we mark the canary as controlled, BINSEC/RSE finds an exploit in about 15 min. This suggests the canary brings a real protection against exploitation.

CVE-2019-14192 in U-boot U-boot is an open-source bootloader, popular for embedded boards. When booting over Network File System (NFS), U-boot does not validate the length field of some network packets. This length is subtracted 16 and used as a size to be copied. If a malicious packet declares a length of less than 16, computation underflows and leads to a buffer overflow.

We encode the situation as follows: the input network packet is controlled, the IP address of the victim is constant, the NFS state machine is initialized to expect the appropriate packet type and all other values are uncontrolled. BINSEC/RSE with the $\text{RSE}_\forall$+ variant (RSE+ times out here) proves in about 2 minutes that a memory copy of more than 4 GB is robustly reachable, which is a strong indication of the criticality of this denial-of-service vulnerability.

CVE-2019-19307 in Mongoose Mongoose is an embedded networking library. When receiving large MQTT packets, the length of the parsed packet can be computed as 0. The parsing loop does not advance and is thus infinite. We look for network packets whose length is parsed as 0 but are accepted as valid. BINSEC/RSE proves in less than a second that such situations are robustly reachable when only the network packet is controlled, confirming exploitability.

CVE-2015-8370 in Grub (aka back to 28) Grub is a bootloader used in most Linux systems. The original vulnerability is an integer underflow leading to buffer underflow when the user types 28 times on backspace on the password prompt of grub. We extracted the vulnerable function, ported it to Linux and simplified it so that it overwrites a local variable instead of the Interrupt Vector Table which is not easily modelled with BINSEC. BINSEC/RSE proves in 17s that the vulnerability is robustly reachable.

4.6.2.2 Flaky tests

Consider the test suite of a program. Ideally, it should fail when the program is incorrect, and succeed when no buggy code path has been exercised. A test is flaky when its outcome is non-deterministic. This is undesirable and line of work [91, 120, 98, 83] with a recent surge in popularity looks into detecting, classifying and reasoning about such tests.

Robust reachability can be used not only to detect flaky tests, but also to choose the inputs to pass to the function to be tested to make the test non-flaky.

Detection. Flakiness can be seen as a special case of non-robustness when labeling non-deterministic inputs as uncontrolled: a test is flaky when the "success" outcome is not robustly reachable.

Actually, the full expressiveness of robust reachability is not necessary to characterize flaky tests, as a test normally has no explicit input, only implicit, uncontrolled inputs. Therefore, the property of interest for a whole test is "for all implicit inputs, success is reached".

Sturdy input generation. Consider Figure 4.8 where we test the functionality of function `foo`. This test is flaky, as line 14 is not robustly reachable. We want to fix the flakiness of this test

Table 4.3: The 46 reachability problems selected for our evaluation

Type	Description		Controlled variable
Real	Vulnerability	CVE-2019-14192 (U-boot)	Network packet
		CVE-2019-20839 (libvncserver)	Socket path
		CVE-2019-19307 (mongoose)	Network packet
		CVE-2019-15900 (doas)	Configuration file
		CVE-2015-8370 (grub, simplified)	Password entry
	CTF	Flare-on 2015 1 & 2	Text entry
		Nintendo Coding Game	Input to hash function to invert
		Manticore	Text entry
	Function inversion	musl (strptime, strverscmp, atoi, strtol) busybox (chmod mode and ip parsing) μclibc (fnmatch) openssl (base64 decoding)	Preimage
Synthetic	Motivating example of [56] and variants		Coefficients to affine function
	Motivating example of [54, Figure 2.2]		Text entry
	SSP bypass	See Section 4.2	Overflowing buffer
	ASLR bypass	2 examples	Various
	Undefined behavior	Overflow flag after 3-bit `shl` in `x86`	None
	Other	Various	Various

```
1  void foo (int x) {
2  ●   if (x % 2 == 0) {
3  ●       return;
4  ●   } else if (!nondet) {
5  ○       error();
6     }
7  ○   return;
8  }
9  int main() {
10    int x = 3;
11    foo(x);
12    x += 2;
13    if (x!=4) { error(); }
14    return 0;
15 }
```

In function `foo`, robustly reachable nodes when `x` is symbolic are marked as ● and non robustly reachable as ○.

Figure 4.8: Example of flaky test adapted from [98]

by finding a value for input `x` of function `foo` that makes the test deterministic. We mark `x` as a controlled, symbolic input and leave `nondet` as uncontrolled input. Success becomes robustly reachable, and BINSEC/RSE even reports that `x` = 2 guarantees deterministic test execution. This allows us to fix our test, and this time, we really used the full power of robust reachability.

Additionally, consider the CFG of function `foo`: line 4 is robustly reachable but not its children lines 5 and 7. This is the sign that it is actually a *source of flakiness*. And indeed, one can modify the conditional line 5 so that all lines in `foo` become robustly reachable.

4.6.3 Experimental evaluation

Research Questions We now seek to investigate in a more systematic way the following research questions:

1. **Precision**: What is the best algorithm for robust reachability in terms of correctness and completeness?
2. **Gain associated to robustness**: Is standard SE subject to false positives and does robust reachability avoid them in practice?
3. **Path pruning**: Does universal path pruning (Section 4.5.4) help explore fewer paths than normal path pruning?
4. **Performance**: What is the overhead of robust reachability?

Protocol We base our analysis on a set of 46 reachability problems on binary executables from various architectures (i686-windows, i686-linux and armv7-linux) presented in Table 4.3. The average trace length for reachable problem instances is 809 instruction-long, with a maximum of 18k instructions. The problems fall into two categories: real code and synthetic examples (*e.g.* code designed to be analyzed). For each executable, BINSEC/RSE determines if a certain location is robustly reachable from a certain initial state. If this is the case a model is output by BINSEC/RSE, and compared to a ground truth obtained by manual analysis. Tests were run on Intel Xeon E-2176M(12)@4.4GHz and we use Z3 4.8.7. Results are classified as follows:

Correct BINSEC/RSE proves the expected result, i.e. it either reports a robust trigger or rightfully proves the absence of such a trigger;

False positive a fragile trigger is reported;

Inconclusive BINSEC/RSE reports no trigger but search was incomplete or the solver returned UNKNOWN at some point;

Resource exhaustion timeout is an hour and memory usage is capped to 7 GB.

Precision (RQ1) As expected, robust variants do not report any false positives, and path merging increases completeness. RSE variants with universal path pruning ($\text{RSE}_\forall$, $\text{RSE}_\forall+$) are less complete than those with existential path pruning, but they are less prone to timeouts, see for example CVE-2019-14192 in U-boot (Section 4.6.2). RBMC suffers from path explosion (time out) much more often than RSE variants. *Overall, Robust SE with path merging and existential path pruning is the most promising method among those presented here, with 44/46 correct answers.* $\text{RSE}_\forall+$ *is less complete but terminates more often.*

Note that two interesting test cases in the "real" category of Table 4.3 need path merging to prove robust reachability: one where a pointer with uncontrolled alignment is passed to `memcpy`, and one where a branch depends on the result of IO. These situations are common programming idioms, demonstrating the importance of path merging.

Table 4.4: Comparison of standard and robust algorithms over our 46 test cases

	SE	BMC	$RSE_\forall$	$RSE_\forall$+	RSE	RSE+	RBMC
Correct	30	22	30	34	37	44	32
False positive	16	14					
Inconclusive			16	11	7		1
Resource exhaustion		10		1	2	2	13
Total time (s)	2725	36911	3947	4374	13590	11534	47784
…w/o resource exhaustion	2725	911	3947	3589	6390	4334	984

RSE: Robust Symbolic Execution. RBMC: Robust Bounded Model Checking. + in acronyms denotes path merging, and ∀ universal path pruning.

Gain associated to robustness (RQ2) We compare standard SE with RSE+, the most precise algorithm of **RQ1**. *Standard reachability has about 30% false positives while robust reachability has none, at the cost of slightly more timeouts.*

There are no false positives in code in the "real" category, except in CVE replays. Our interpretation is that well-functioning programs are designed to behave the same regardless of the uncontrolled environment: concrete memory layout, stack canaries, *etc.* Robust reachability becomes decisive on buggy code, notably with undefined behavior. This is also illustrated by case studies (Section 4.6.2).

Path pruning (RQ3) We compare $RSE_\forall$, which features universal path pruning, to RSE, which features usual path pruning. Comparison is limited to test runs of more than a second which succeed with both methods. This is to prevent comparing a run where Binsec/RSE proves that the target is reachable and stops, to a run where Binsec/RSE does not find the target and explores the whole program. *$RSE_\forall$ explores 17% less paths and interprets 21% less instructions than RSE.* This comes at the price of more universally quantified SMT queries: the average time per SMT query goes up by 25%. Overall the run time of both methods is very close.

With path merging, the difference in paths explored disappears: $RSE_\forall$+ explores 1% less paths and instructions than RSE+. This is due to the fact that for some tests, path merging "unlocks" some new paths. Overall, $RSE_\forall$+ is 6% slower than RSE+ on successful, terminating tests.

Performance (RQ4) In this question, we compare the run time of robust algorithms to SE. Comparison is done on the same basis as before, except that we count timeouts. RSE+ is 74% slower than standard SE on geometric average. This is mostly due to newly introduced time-outs (up to 260× slower) since median slowdown is only 15%. $RSE_\forall$ is more consistently slower with about 30% slowdown in both geomean and median. This is mainly explained by increased solver time (universal path pruning queries). $RSE_\forall$+ is close in median slowdown, but path merging introduces new timeouts and drives the average slowdown up to 62%. *RSE+ has a low overhead compared to standard SE, except for a few time-outs (2/46).*

4.6.4 Additional considerations

4.6.4.1 Going beyond reachability

The formal framework we provide in Section 4.4.1 allows to lift any reachability property to its robust equivalent. Could we do the same for other classes or properties and hyperproperties? Formally, yes, but we may need to give up part of the results of this chapter, and in the case of hyperproperties, the lifted property might even be nonsensical, so some care must be taken.

Robust trace properties A trace property is a set of traces $\Pi \subseteq \mathcal{S}^\infty$. A program P satisfies Π if $T(P) \subseteq \Pi$. One can apply the same construction as in Definition 14 to obtain a lifted "robust" property $\mathfrak{R}(\Pi)$: P satisfies $\mathfrak{R}(\Pi)$ if $\exists a.\, \forall x.\ P|_{(a,x)} \vdash \Pi$.

For example, consider non-termination. Robust non-termination expresses that for some controlled input, the program is guaranteed not to terminate. In a security context, this encodes a form of "guaranteed denial-of-service".

A well studied class of trace properties is the class of *safety properties*. They are the negations of the reachability properties as defined in Definition 1, or described more intuitively, they are trace properties that can be falsified by observing a bad finite trace prefix. As an example of safety property, consider the absence of null pointer dereference. Lifting it like before, "robust absence of null pointer dereference" expresses that for some controlled input, the program is guaranteed to be free of null pointer dereference. This property is weaker that the original one, and makes little sense with the threat model we used until now: why would we rely on the attacker to establish safety of our program? We actually need to reverse this threat model: consider the controlled input as controlled by the system administrator trying to harden the system. Then the property consists in looking for a system configuration which is impervious to attacks.

While the construction we introduced still works for trace properties, the proof methods of Section 4.5 do not. Developing algorithms to prove robust trace properties is left to future work.

Robust hyperproperties Informally, robustness lifts a trace property Π by adding quantification over the inputs of the system, therefore a quantification over traces: there must be a controlled input, such that all traces starting with this input satisfy Π. But since hyperproperties (introduced in Section 4.4.3) are also allowed to quantify over traces, the quantifications might collide. Consider the hyperproperty Π "the program terminates on average in less than 100 steps". Lifting Π to robustness would yield "there exists a controlled input, such that for all uncontrolled inputs, the only trace starting with those inputs terminates in average in less than 100 steps." Obviously, "average" here lost its meaning.

We can solve this issue by generalizing the construction of Definition 14. We now split inputs to the program into three parts: controlled inputs a, uncontrolled input x, and remaining input y. The lifted hyperproperty only quantifies on y. In the example above, Q becomes "there exists a such that for all x, the system terminates in less than 100 steps on average on y".

Formally, we need to adapt the restriction of a program P to only partial inputs: $P|_{(a,x,\cdot)} = \{t \in P \mid \exists y.\, t_1 = s_1(a, x, y)\}$. Then, lifting the hyperproperty Π yields:

$$\{P \subseteq \mathcal{S}^\infty \mid \exists a.\, \forall x.\ P|_{(a,x,\cdot)} \vdash \Pi\}$$

Whether this construct has a useful meaning is very context-dependent. But let us give an example for non-interference, or rather its negation. Consider the case where the attacker is only one of many unprivileged users, and has the goal of breaking non-interference, *i.e.* observing low outputs that leak information on the inputs of privileged users. We label as controlled the inputs

a of the attacker, and denote as x and y the inputs of other unprivileged and privileged users respectively. If $(a, x, y) \sim (a', x', y')$ denote that traces starting with inputs (a, x, y) and (a', x', y') are observationally equivalent (be it termination-sensitive or not, for the sake of simplicity), then one would write non-interference as:

$$\forall a, x, y,' y. (a, x, y) \sim (a, x, y')$$

Robust violation of non-interference is then

$$\exists a. \forall x. \exists y, y'. (a, x, y) \not\sim (a, x, y')$$

which means that the attacker can choose wisely a controlled input a such that, whatever other unprivileged users do, a leak of information on high input y will happen. Here we only let non-interference quantify over y.

4.6.4.2 Negation of robust reachability

We focus in this chapter on (positively) proving robust reachability and discussing the potential applications for security assessment. Let us briefly discuss now the case of proving the *negation* of robust reachability. This property, which falls in the category of *collaborative invariance* [1], expresses that for all controlled inputs a, there exists an uncontrolled input x that prevents some event O. In other words, it is always possible (for the system, for the defender, *etc.*) to preserve the invariant $\neg O$. While this is weaker than $\neg O$, it is still relevant for security as it characterizes those systems that may not be fully secure (the invariant does not hold) but which can still always be defended. From a broader perspective, this can be an interesting step toward a principled definition of soundiness [90]: instead of discarding a system because it does not uphold the expected invariant, we can still show that it can be made to work, and thus see more finely the value we can attach to it.

4.6.4.3 Scope of the definition

We excluded interactive systems and quantitative approaches from our definition of robustness (Definition 14, Section 4.4.1) to keep automated proof methods tractable. A quantitative approach will be discussed at length in Chapter 5. We notably show in Section 5.9.2 (**RQ4**) that a quantitative counterpart to satisfiability of universally quantified formulas would be about 7 times slower with additional time-outs.

Remains the question of interactive systems. In this section we argue experimentally that handling them would yield significant overhead.

Assume we want to model a leak in ASLR in libvncserver (Section 4.6.2): the attacker knows about an address z and wants to use the bug to jump to z. The corresponding property is: for all values[2] of z, there exists an attacker input a such that for all other uncontrolled inputs x, control flow is diverted to z. This uses another universal quantifier, which we exclude in our definition of robust reachability (Section 4.4.1) to keep satisfiability queries tractable. Similarly, in our case study on doas, we would like to check that the exploit works for any typoed username and, and for any user ID and group ID. These are two example situations with only one half additional round-trip of interaction between the system and the attacker, which we use as a lower-bound of the cost of supporting fully interactive systems.

We implemented a naive encoding of this into SMT formulas with one additional quantifier, and in both cases, RSE+ does not terminate within 24h.

[2]Without a null byte, but we ignore this detail for the sake of simplicity.

This is not a scaling issue but a more fundamental one with additional quantifier alternations: none of Z3 [48], Boolector [97] and CVC4 [11] are able to prove in less than 1h that $\forall z.\, \exists a.\, a \text{ XOR } 1 = z$ holds, with 32-bit bitvectors (where the quantification of x is even omitted). On such formulas, the model that the solver attempts to compute to instantiate the existential quantifier is actually a function of z, significantly increasing the size of the search space.

4.7 Related work

Broadly speaking, we are interested in defining a subclass of *comparatively more interesting bugs* amenable to automation. We review related prior attempts.

Automatic exploit generation (AEG) These approaches seek to demonstrate the *impact* of a bug by automatically generating an exploit from it [4, 24, 72]. *This is complementary to robustness*, which focuses on replicability. Actually, both techniques could be advantageously combined, as a replicable exploit is clearly more threatening than a fragile one. Current AEG methods being based on symbolic methods, adapting them for robustness looks feasible.

Quantitative reasoning & model counting Several approaches rely on probabilities or counting to distinguish important issues from minor ones — for example (quantitative) *probabilistic model checking* [70, 5] or *quantitative information flow analysis* [73]. Robust reachability could be refined in such a way. Yet, current quantitative approaches do not scale on software, as they often rely either on the finite-state hypothesis, or on *model counting* solvers [67], which are only at their beginning (see Sections 4.4.1 and 4.6.4.3).

Fairness *Fairness assumptions* in model checking [71] aim at discarding traces considered as unrealistic and avoiding false alarms from the user point of view. While the goal is rather similar to ours, the two techniques are very different: fairness assumptions typically require certain sets of states to be visited infinitely often along a trace, while robust reachability requires that a trace cannot be influenced by uncontrolled input w.r.t. a given reachability property.

Symbolic Execution and quantifiers Finally, while symbolic execution is commonly performed with quantifier-free constraints, a notable exception is *higher-order test generation* [62], where Godefroid proposes to rely on universally quantified uninterpreted functions ($\forall\exists$ queries) in order to soundly approximate opaque code constructs. Higher-order test generation and robust reachability are complementary as they serve two different purposes: robust reachability can only be used in a modest way for opaque code constructs (finding controlled inputs for which their value does not matter), while higher-order test generation is inadequate for robust reachability, as it would be as if the attacker could choose the controlled inputs knowing the uncontrolled ones.

4.8 Conclusion

We introduce the novel concept of robust reachability, that we argue is better suited than standard reachability in several important scenarios for both security (e.g., criticality assessment, bug prioritization) and software engineering (e.g., replicable test suites). We formally define and study robust reachability, discuss how standard symbolic methods to prove reachability can be revisited to deal with the robust case, design and implement the first robust symbolic execution

engine and demonstrate its abilities in criticality assessment over 5 CVEs. We believe robust reachability is an important sweet spot in terms of expressiveness and tractability. We hope this first step will pave the way to more refinements and applications of robust reachability.

Chapter 5

Quantitative robustness

In this chapter, we acknowledge how frustrating it is that robust reachability dismisses bugs which an attacker can trigger 99% of the time. We will thus resort to model-counting-like approaches to compute this actual number of 99%. We define this quantity formally, and investigate possible proof methods. An extensive comparison of algorithms from various fields leads us to introduce a new, approximate one, leading to encouraging results in vulnerability assessment case studies.

5.1 Introduction

Context & Problem In the previous chapter, we developed *robust reachability* to determine whether an attacker can reproduce a bug reliably: a bug is robustly reachable if an attacker can choose the part of input he controls so that for all values of the rest of inputs, the bug is triggered. This is meant to be a sign of the security relevance of the bug. However, while standard reachability is too weak and may lead to practical false positives, robust reachability may be too strict a notion in many cases. Robust reachability requires that when the attacker plays optimally, the bug is triggered 100% of the time. Naturally, we would also want to detect bugs which only happen 99% of the time, while still dismissing those which happen for one input out of 10^{30} at best. Both are reachable but not robustly reachable. What we need is a sort of *quantitative* counterpart to robust reachability.

Goal and challenges We want to design a formal definition of 99% in the kind of statements we made above. This sounds like model counting in the sense that we count inputs that trigger the bug, but we additionally want to take the presence of the attacker into account like robust reachability does: attacker input is chosen as worst case, and other input is counted. A value of 0 should correspond to unreachable bugs, and the maximal value of 100% should correspond to robustly reachable bugs. With such properties, a quantitative approach would give us access to all the intermediate, fractional situations that standard reachability and robust reachability would fail to characterize.

Proposal Like with robust reachability we split the program input into attacker-controlled input a and uncontrolled input x. We define *quantitative robustness* as the proportion of uncontrolled inputs x which trigger the bug when the attacker chooses controlled input a optimally. This problem looks very similar to model counting, however the two notions are distinct. If f is a

function of (a, x) expressing that the bug is hit, model counting yields $|\{(a, x) \mid f(a, x)\}|$ while we want $\max_a |\{x \mid f(a, x)\}|$, normalized between 0 and 1. It turns out that the problem of computing $\max_a |\{x \mid f(a, x)\}|$ for f a propositional formula is known as functional **E-MAJSAT**[1] [87] (f**-E-MAJSAT** for short) and has been studied under varying names for Bayesian networks inference and probabilistic planning notably. It is a hard problem where solvers [75, 101, 85, 92, 57] are often tuned for specific kinds of instances. As we will see, improvements of f**-E-MAJSAT** solvers for probabilistic planning are not always beneficial for our instances coming from program analysis, and we will end up modifying two existing techniques to better fit this new domain of application.

Contributions We claim the following contributions:

- We define a quantitative pendant of robust reachability called quantitative robustness (Section 5.4), and we propose Quantitative Robust Symbolic Execution (QRSE), a variant of symbolic execution to determine the quantitative robustness of the target, provided that one can solve f**-E-MAJSAT** problem instances (Section 5.5);
- We present a number of existing works about f**-E-MAJSAT** and the many variants that have been studied in various fields like Bayesian networks and probabilistic planning, but which are not well known in the domain of program analysis (Sections 5.6 and 5.7);
- We introduce a novel parametric algorithm to solve f**-E-MAJSAT** where one can tune the trade-off precision *vs.* performance by a technique we call *relaxation* (Section 5.8). Extreme values of the parameter degenerate into already known techniques;
- We evaluate the ability of these techniques to solve the f**-E-MAJSAT** problems stemming from quantitative robustness computation, which reveals that techniques that fare well for probabilistic planning for example become detrimental for our purposes, and justifies the need for relaxation (Section 5.9.2). We also illustrate the usage of quantitative robustness through Quantitative Robust Symbolic Execution (QRSE) on realistic case studies of vulnerability assessment (Section 5.9.3).

Quantitative robustness is a new compromise to assess the replicability of a bug: it is more precise, but more expensive than standard reachability and robust reachability.

5.2 Motivating example

Loosely inspired by CVE-2019-15900, consider in Figure 5.1 the case of two programs incorrectly using initial memory to determine the privileges of the attacker. Specifically we consider a network server performing a command with an argument on behalf of the attacker. Whether the attacker can perform sensitive commands depends on a `privilege_level` which is accessed through a getter `get_privilege_level` and a setter `set_privilege_level`. We want to consider the consequences of a bug where `get_privilege_level` incorrectly returns uninitialized memory modeled as random garbage.

We want to compare two versions of the server: `prog1`, which accepts two commands `GET_VERSION` which does not do anything useful here, and `SUDO` which allows a user with privilege `OPERATOR_LEVEL` to escalate to the higher `ADMIN_LEVEL`; and `prog2`, which accepts `GET_VERSION` again and

[1] This complex name comes from: MAJSAT is when a MAJority of x satisfy the formula, E-MAJSAT when there Exists a a such that MAJSAT. E-MAJSAT is a decision problem, and *functional* E-MAJSAT is the corresponding functional problem where one wants to compute the value instead of comparing it to 1/2.

```c
/* main privilege levels */
#define DEFAULT_PRIVILEGE_LEVEL 1
#define OPERATOR_LEVEL 100
#define ADMIN_LEVEL 9000
/* commands */
#define DROP_PRIVILEGE 0
#define DROP_PRIVILEGE_LEGACY 1
#define GET_VERSION 2
#define SUDO 3

uint32_t uninitialized; // random garbage
uint32_t privilege_level = DEFAULT_LEVEL;

void set_privilege_level(uint32_t new) {
  privilege_level = new;
}

uint32_t get_privilege_level() {
  // bug: return uninitialized memory
  return uninitialized;
}
```

```c
void prog1(uint32_t command, uint32_t argument) {
  if (command == GET_VERSION) {
    /* harmless */
  } else {
    /* command is sudo */
    if (get_privilege_level() == OPERATOR_LEVEL) {
      set_privilege_level(ADMIN_LEVEL);
    }
  }
}

void prog2(uint32_t command, uint32_t argument) {
  switch (command) {
    case GET_VERSION: /*harmless*/ break;
    case DROP_PRIVILEGE: case DROP_PRIVILEGE_LEGACY:
      if (argument<get_privilege_level()) {
        set_privilege_level(argument);
      }
  }
}
```

Figure 5.1: `prog1` and `prog2` are both vulnerable, but one is more than the other

`DROP_PRIVILEGE` (which has two distinct codes, for example for backward compatibility) which allows a user to reduce its privilege level to a chosen lower value.

Is it possible that the attacker obtains privilege level greater or equal to `ADMIN_LEVEL` by submitting a carefully chosen command and argument to these functions? For `prog1`, the attacker obtains admin privilege when `command` is not 2 and `get_privilege_level` returns 100. For `prog2`, the attacker obtains admin privilege when `command` is 0 or 1, the asked privilege level `argument` is both over 9000 (`ADMIN_LEVEL`) and below `get_privilege_level()`. Formally, this corresponds to the following respective exploitation conditions:

$$f_1 \triangleq \texttt{command} \neq 2 \wedge \texttt{uninitialized} = 100$$

and

$$f_2 \triangleq \texttt{command} \in \{0, 1\} \wedge 9000 \leq \texttt{argument} < \texttt{uninitialized}$$

In `prog1`, when the attacker plays perfectly by choosing `command` $= 1$, he needs to have a lot of luck: only one value of `uninitialized` (100) out of 2^{32} lets him win. To the contrary, in `prog2`, for `command` $= 1$ and `argument` $= 9000$, a large majority of values of `uninitialized` will let the attacker achieve his goal. We want to develop an automated machinery to back this intuition.

Qualitative methods Traditional bug finding techniques are of little use here: they prove that the attack is *reachable*, *i.e.* that formulas f_1 and f_2 admit both at least one solution. We can refine: robust reachability (Chapter 4) states that the attack always works when the attacker plays perfectly: $\exists \texttt{command}, \texttt{argument}. \forall \texttt{uninitialized}. f_x$, but this is too strict here and neither program satisfies it.

Model counting Where these *qualitative* techniques fail to distinguish our two programs, maybe a more *quantitative* one will bear fruit. For example, we could compare the number of solutions of f_1 and f_2, or rather their density in a search space of size 2^{96}. This is reminiscent of probabilistic symbolic execution [59]. For f_1, this density is $\frac{(2^{32}-1)\times 2^{32}\times 1}{2^{96}} \simeq 2.3 \cdot 10^{-10}$. For f_2 the computation is slightly more cumbersome. Valid pairs $(\texttt{command}, \texttt{uninitialized})$ are

counted as follows:

$$\sum_{u=9000}^{2^{32}-1} u - 9000 = \sum_{v=0}^{2^{32}-9001} v = \frac{1}{2}(2^{32} - 9001)(2^{32} - 9000)$$

and the density of solutions of f_2 is thus $\frac{2(2^{32}-9001)(2^{32}-9000)}{2\times 2^{96}} \simeq 2.3 \cdot 10^{-10}$. The density of f_1 and f_2 are in fact extremely close, and worse, they compare in order opposite to what we expect: $f_1 > f_2$.

Our approach The missing ingredient is taking into account the threat model: the attacker will choose the best possible input he can, *i.e.* `command = 1` and `argument` $= 9000$, but he cannot influence the value of `uninitialized`. What we want to compute is the amount of solutions for the value of `command` and `argument` most favorable to the attacker:

$$\max_{\texttt{command,argument}} |\{\texttt{uninitialized} \mid f_1\}| = |\{100\}| = 1 \tag{5.1}$$

$$\max_{\texttt{command,argument}} |\{\texttt{uninitialized} \mid f_2\}| = |[9001; 2^{32} - 1]| = 2^{32} - 9001 \tag{5.2}$$

These numbers can be fairly compared as the search space has the same size (2^{32}) but in the general case we will consider a proportion of inputs instead, which we call *quantitative robustness*. Quantitative robustness does align to the intuition we had: it is low ($2.3 \cdot 10^{-10}$) for `prog1` but close to 1 for `prog2`: approximately 0.9999979043.

The problem of doing computations like eqs. (5.1) and (5.2) on a boolean formula is known as functional **E-MAJSAT** [87], or **f-E-MAJSAT** for short. Solvers exist for this problem or rather close variants because most of the literature is targeted to other domains like Bayesian network inference [76, 122] or probabilistic planning [75, 101]. f_1 is small and two algorithms of the literature are able to obtain eq. (5.1) (DC-SSAT [92] and an unnamed algorithm mentioned in Huang [75, Algorithm 1, p. 257] which we call CONSTRAINED) in less than 2 seconds. On the other hand, we know of no existing **f-E-MAJSAT** solver able to obtain eq. (5.2) in a reasonable time (we tested CONSTRAINED, SSATABC [85], DC-SSAT, and COMPLAN+ [101] with a timeout of 20 minutes). If we turn to approximate solutions, MAXCOUNT [57] also times out, and an upper bound taken from COMPLAN+ terminates quickly but without a useful result: it proves that the quantitative robustness is upper-bounded by 1, which is a tautology. In other words, even for this simplistic example, existing techniques come short, presumably because they are tuned for instances of their respective original domain. Taking inspiration from existing knowledge-compilation based algorithms, we propose a technique called relaxation that offers an interesting trade-off between performance and precision. For `prog2` we obtain[2] in about 1 second that the quantitative robustness of privilege escalation is comprised between 0.999997904291 and 1. This is enough to conclude that there are many more initial states that let the attacker exploit the vulnerability in `prog2` than in `prog1`. We interpret this as a sign that this bug is presumably more severe in `prog2` than in `prog1`.

Discussion We are counting models without assigning a weight, or rather a probability, to each of them. In effect, this is equivalent to arbitrarily assigning a uniform distribution to uncontrolled inputs. It would feel more natural to be able to assign any probability distribution to uncontrolled inputs, to get an even more precise result. We do not do so in our experiments for two reasons.

[2] With parameters BFS(40) and precise lower bound.

Firstly, assigning a meaningful probability distribution to uncontrolled inputs is often quite hard. Uncontrolled inputs quite often represent implicit inputs to the program, like the initial content of memory, registers, *etc.* In the threat model of robust reachability, the attacker does not know the value of uncontrolled inputs; they should thus be values that he cannot predict or describe. It is not uncommon that it is also the case for the defenders: for example, how would you describe the distribution of the pointer returned by `malloc`? Besides, when we can describe the actual distribution, it is often that randomness was intentionally introduced: stack canaries are uniformly distributed, ASLR adds well known entropy to pointers, and hash function return values indistinguishable from random. In these cases, the uniform distribution on intervals is actually enough. The case of uninitialized memory is interesting in this regard. On non-embedded systems, initial memory given by the OS is zeroed, so deterministic. Memory reused from the stack or the heap is however not zeroed, and its content could in theory be predicted by further program analysis. In practice, one can thus either analyze the full program including code before `main` starting from zeroed memory, which is most probably intractable, or do an underconstrained (in the sense of Ramos and Engler [102]) analysis by abstracting initial memory as "anything, really" to eschew the cost, in which case we purposefully ignore the complexity of the actual distribution of inputs, and we might as well choose the uniform one.

Secondly, on a practical note, none of the techniques and tools we will consider in Section 5.7 support assigning arbitrary probability distributions to inputs. Most of them support assigning independent Bernoulli distributions to individual input bits, but cannot handle dependent bits. We focus on a common denominator in terms of expressiveness.

Finally, let us reiterate that the goal is not to give a definitive rational value of vulnerability, but rather to give a hint to defenders. We want to be able to distinguish between highly improbable situations and quite probable ones. The uniform distribution is a good enough approximation of moderately skewed ones for this purpose.

5.3 Background & Notations

We reuse the notations of Section 3.2. Notably for a formula f, the set of its models is $M(f)$, its model count is $\sharp(f)$, the set of variables effectively appearing in f is $V(f)$, and for a partial valuation $a \in \mathbb{B}^A$, $f|_a$ is the formula identical to f but where variables in A are replaced by their image by a. We recall that model counting is then the following problem:

Definition 4 (model counting)**.** ♯SAT is the following function problem: given a propositional formula f in CNF, output $\sharp(f)$

f-E-MAJSAT can be seen as a generalization of model counting, and some techniques to solve it are close to model counting algorithms. In this background section we will detail a specific one which will come useful later.

5.3.1 Normal forms for model counting

To solve model counting one can use formulas in a specific normal form: deterministic Decomposable Negational Normal Form (d-DNNF) [41]. We will in fact use, w.l.o.g., a slightly stricter normal form: decision Decomposable Negational Normal Form (decision-DNNF) [53].

Definition 17 (decision-DNNF)**.** A formula in decision Decomposable Negational Normal Form (decision-DNNF) is a Directed Acyclic Graph (DAG) of the following nodes:

True and False nodes $\top$ and $\bot$;

decomposable And node $\bigwedge_{i=1}^{n} f_i$, where $\forall 1 \leq i, j \leq n. V(f_i) \cap V(f_j) = \varnothing$, and the children $(f_i)_{1 \leq i \leq n}$ are in decision-DNNF;

Decision (or Ite) node $\text{ite}(v, f, g)$, where f and g denote formulas in decision-DNNF, v a variable, and $v \notin V(f)$, $v \notin V(g)$.

An example is given in Figure 5.2. By convention, $V(\top) = V(\bot) = \varnothing$, $\sharp(\top) = 1$, $\sharp(\bot) = 0$. This definition is slightly non-standard: literals are normally included, but we replace v by $\text{ite}(v, \top, \bot)$ and $\neg v$ by $\text{ite}(v, \bot, \top)$.

The process of converting a CNF formula to an equivalent decision-DNNF formula is called decision-DNNF-compilation. D4 [82] is a decision-DNNF compiler. d-DNNF-compilers are more common, but interestingly, while d-DNNF compilers like C2D [42] and Dsharp [94] officially output d-DNNF, they actually produce the stricter decision-DNNF. This is because they use algorithms inspired by DPLL search where decisions naturally lead to the ite nodes of decision-DNNF.

In other words, a large part of the literature handles formulas in d-DNNF, but actual implementations produce the stricter decision-DNNF. The model counting algorithm presented in this background (Theorem 1) is originally given for d-DNNF, but we present it lifted to decision-DNNF.

About smoothness One cannot compare the model count of formulas with different sets of variables, which becomes cumbersome when trying to compute the model count of a formula from the model counts of several subformulas. To simplify formalism, one usually resorts to the notion of smooth formula.

Definition 18 (Smoothness [43])**.** A formula in decision-DNNF is said to be smooth if all ite nodes $\text{ite}(v, f, g)$ verify $V(f) = V(g)$.

Crucially, for smooth formulas $\sharp(\text{ite}(v, f, g)) = \sharp(f) + \sharp(g)$. Without it, one must reason about pairs $(\sharp(f), V(f))$ instead of just $\sharp(f)$ which makes the formal treatment considerably heavier. As usual in the literature, we present the formalism on smooth formulas only, which can be done without loss of generality [43] as a formula can be made smooth in polynomial time.

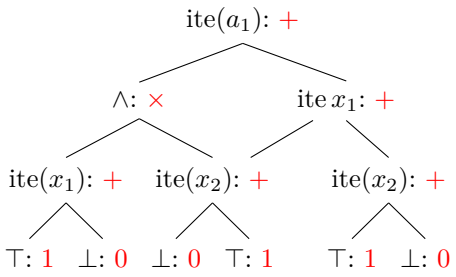
5.3.2 Basic algorithms for model counting

One can count the number of models of a smooth decision-DNNF formula in linear time:

Theorem 1 (model counting of decision-DNNF [53])**.** *The model count of a formula in smooth decision-DNNF can be computed as follows:*

$$\begin{aligned} \sharp(\bot) &= 0 \\ \sharp(\top) &= 1 \\ \sharp(\text{ite}(v, f, g)) &= \sharp(f) + \sharp(g) \\ \sharp\left(\bigwedge_{i=1}^{n} f_i\right) &= \prod_{i=1}^{n} \sharp(f_i) \end{aligned}$$

In other words, And nodes correspond to multiplication of model counts, and Ite nodes to addition, as illustrated in Figure 5.2.



For $f = \text{ite}(a_1, x_1 \wedge \neg x_2, \text{ite}(x_1, \neg x_2, x_2))$, Theorem 1 yields $\sharp(f) = (1+0) \times (0+1) + (0+1) + (1+0) = 3$.

Figure 5.2: A formula in decision-DNNF (black), and model counting for it (red).

Compilation This algorithm reduces the problem of model counting to compiling the original CNF formula to an equivalent decision-DNNF one.

The complexity of compilers like c2d [42] which rely on a DPLL-like search composed of decisions and then propagations is given as exponential in the treewidth of the formula. Let us briefly define this notion. One considers the connectivity graph of the formula: the graph where every variable is a node, and there is an edge from v to w when there is a clause where both v and w appear. We consider the process of removing nodes one by one in some variable order x. The maximal degree of a node at removal time during this process is called width of the variable order x. The minimal width of all possible orders is called the treewidth of the formula [107]. The link with DPLL-like algorithms is that the number of neighbor nodes at removal time corresponds to the number of variables that must be decided before the node can be eliminated by unit propagation. This traduces the fact that choosing the right variable order for decisions in compilers is of paramount importance for performance. We will later talk about restricting possible orders: this increases the treewidth of the formula, and thus impacts performance of the compilation.

As compilation can increase the size of the formula exponentially, the linear time complexity of Theorem 1 does not mean that model counting on CNF formulas is efficient. In practice the complexity of model counting resides in decision-DNNF-compilation. Model counting on the decision-DNNF accounts for a comparatively low amount of work (less than 4% on the corpus of our experiments of Section 5.9.2).

Conditioning For a partial valuation $a \in \mathbb{B}^A$ and a formula f in decision-DNNF it is possible to compute a formula equivalent to $f|_a$ also in decision-DNNF as follows: replace $\text{ite}(v, g, h)$ by g if $v \in A$ and $a(v) = \top$, h if $v \in A$ and $a(v) = \bot$ and otherwise leave it as is. Thus, we can compute $\sharp(f|_a)$ in linear time as well.

5.3.3 Beyond model counting

As mentioned earlier, to characterize how reliably an attacker can trigger a bug, we do not precisely want to solve model counting but an extension of it called functional **E-MAJSAT**. We will make a more complete panorama of such extensions of model counting later in Section 5.6 and of the corresponding solving algorithms in Section 5.7, but let us define **f-E-MAJSAT** now.

Definition 19 (**f-E-MAJSAT** [87])**.** **f-E-MAJSAT** is the following function problem:

Input a formula f in CNF with a partition of variables in A and X: $V(f) = A \uplus X$.

Output $\max_{a_1,\ldots,a_n \in \mathbb{B}^A} \sharp\left(f|_{a_1,\ldots,a_n}\right)$

As usual with functional problems, there is a companion decision problem called **E-MAJSAT** which tests whether f**-E-MAJSAT** is above $2^{|X|-1}$ (or another threshold).

Variables in A are called *choice variables* and variables in X are called *chance variables*. The distinction between chance and choice variables the key to encode the presence of the attacker and the partition of inputs into controlled and uncontrolled inputs.

f**-E-MAJSAT** reduces to **SAT** when $X = \varnothing$ and to $\sharp$**SAT** when $A = \varnothing$, so it is at least as hard as these problems.

5.4 Quantitative robustness

In this section, we define quantitative robustness and consider the nature of the corresponding computational problem.

5.4.1 Formal definition

We consider the same threat model as for robust reachability (Chapter 4) where an attacker can choose *controlled inputs* $a \in \mathcal{A}$ and other inputs to the program are unknown to the attacker and called *uncontrolled inputs* $x \in \mathcal{X}$. We also reuse the same notations: the program P to be analyzed is represented a transition system with transition relation $\rightarrow$ over the set of states $\mathcal{S}$. A trace is a succession of states respecting $\rightarrow$; the set of traces of a program P is $T(P)$. $P|_{(a,x)}$ is the program identical to P but executed on input (a, x). We say that a program P can reach a set of traces O when $T(P) \cap O \neq \varnothing$, meaning that P admits a trace reaching the goal, and we say that P reaches O robustly when $\exists a \in \mathcal{A}. \forall x \in \mathcal{X}. T\left(P|_{(a,x)}\right) \cap O \neq \varnothing$, meaning that for some controlled input a, for all uncontrolled inputs x, the target is reached.

Definition 20 (Quantitative robustness)**.** We consider the reachability problem associated to program P and target set of paths O. The associated quantitative robustness is

$$q(P, O) \triangleq \frac{1}{|\mathcal{X}|} \max_{a \in \mathcal{A}} \left|\left\{x \in \mathcal{X} \mid T\left(P|_{(a,x)}\right) \cap O \neq \varnothing\right\}\right|$$

It is a rational number between 0 and 1 and corresponds to the maximal proportion of uncontrolled inputs that reaches the target, for the best controlled input.

One can envision at least two ways to use quantitative robustness. First one can make quantitative robustness a decision problem by comparing it to a threshold. For example, this is enough to filter detected bugs, by removing bugs below a low threshold, or by prioritizing bugs above a high threshold. Second we can use the value on its own, as an indication for further manual analysis, or to actually sort bugs by descending robustness. We focus on the latter approach, which is more general than the former.

Scope & limitations This definition inherits some of the limitations of robust reachability. Notably, the attacker can only submit one input to the system, in one go, and without knowledge of uncontrolled inputs. This effectively forbids interactive systems. It would be possible to tailor a formal definition to more interactivity, but this would complicate computing it.

It also has some additional limitations related to the fact that we use model counting: inputs are assumed to be *in finite number* and *uniformly distributed*. It would be possible to use another definition: the maximal (over controlled input) probability of uncontrolled input to reaching the target. This can accommodate infinite input with an appropriate probability measure over $\mathcal{X}$, and non-uniform probability distributions. We avoid this solution:

- Finiteness of input is not very problematic: formally, the state of a computer is finite due to finite registers and memory, and in practice even more so.
- On the other hand, one can believe non-uniform distribution of inputs would be handy. In practice, it would be hard to use because determining the probability distribution of uncontrolled inputs is far from trivial, as we argued in the introduction. Uncontrolled inputs are usually implicit inputs, and they are often unknown not only by the attacker, as the threat model requires, but also to us. Besides, none of the algorithms we will consider to compute quantitative robustness (Section 5.7) support arbitrary distributions. Some, but not all, support more than merely the uniform distribution (notably independent Bernoulli bits), so we chose a sort of common denominator. This ensures maximal tractability at the expense of expressiveness.

5.4.2 Interesting properties

Extrema Extreme values of quantitative robustness correspond to the properties we are already familiar with:

Proposition 10. *Quantitative robustness is 0 if and only if the target is not reachable. Quantitative robustness is 1 if and only if the target is robustly reachable.*

The promise of quantitative robustness is that we now have a criterion to single out bugs which are nearly robust, but not exactly because for few uncontrolled inputs the target is missed: they should have a quantitative robustness close to 1.

Paths Robust reachability can be lost when there is a branch depending on uncontrolled input and recovered later when paths meet again. This forces us to merge paths together. On the other hand, quantitative robustness is not fully lost when paths separate. We denote the restriction of P to paths $\pi_1, \ldots, \pi_n$ as $P|^{\pi_1,\ldots,\pi_n}$, and we start with some properties of quantitative robustness of such a restriction.

Proposition 11 (Monotonicity of quantitative robustness of paths)**.** *Let π be a path in a program P. $q\left(P|^{\pi}, O\right) \leq q\left(P, O\right)$.*

Proof. This comes from the fact that for all $a \in \mathcal{A}$

$$\left\{x \in \mathcal{X} \mid T\left(P|^{\pi}|_{(a,x)}\right) \cap O \neq \varnothing\right\} \leq \left\{x \in \mathcal{X} \mid T\left(P|_{(a,x)}\right) \cap O \neq \varnothing\right\}$$

□

Proposition 12 (Quantitative robustness of merged paths)**.** *Let π, π' be two paths in a program P. Then*

$$q\left(P|^{\pi,\pi'}, O\right) \leq q\left(P|^{\pi}, O\right) + q\left(P|^{\pi'}, O\right)$$

Proof. Let a reaching the max in the definition of $q\left(P|^{\pi,\pi'}, O\right)$.

$$\left\{x \in \mathcal{X} \mid T\left(P|^{\pi,\pi'}\Big|_{(a,x)}\right) \cap O \neq \varnothing\right\} = \left\{x \in \mathcal{X} \mid T\left(P|^{\pi}|_{(a,x)}\right) \cap O \neq \varnothing\right\} \cup \left\{x \in \mathcal{X} \mid T\left(P|^{\pi'}\Big|_{(a,x)}\right) \cap O \neq \varnothing\right\}$$

In terms of cardinal:

$$|\mathcal{X}|q\left(P|^{\pi,\pi'}, O\right) \leq \left|\left\{x \in \mathcal{X} \mid T\left(P|^{\pi}|_{(a,x)}\right) \cap O \neq \varnothing\right\}\right| + \left|\left\{x \in \mathcal{X} \mid T\left(P|^{\pi'}\Big|_{(a,x)}\right) \cap O \neq \varnothing\right\}\right|$$

By definition of quantitative robustness,

$$\left|\left\{x \in \mathcal{X} \mid T\left(P|^{\pi}|_{(a,x)}\right) \cap O \neq \varnothing\right\}\right| \leq |\mathcal{X}|q\left(P|^{\pi}, O\right)$$

hence the result. □

As a result, we can prove that quantitative robustness cannot vanish at a branch:

Proposition 13 (Quantitative robustness pseudo-conservation)**.** *Let $\pi_1, \ldots, \pi_n$ be paths in a program P. There exists $1 \leq i \leq n$ such that $q\left(P|^{\pi_i}, O\right) \geq \frac{1}{n} q\left(P|^{\pi_1,\ldots,\pi_n}, O\right)$.*

Proof. By contradiction, if $\forall 1 \leq i \leq n.\, q\left(P|^{\pi_i}, O\right) < \frac{1}{n} q\left(P|^{\pi_1,\ldots,\pi_n}, O\right)$ then by Proposition 12, we get $q\left(P|^{\pi_1,\ldots,\pi_n}, O\right) < n \times \frac{1}{n} q\left(P|^{\pi_1,\ldots,\pi_n}, O\right)$ which is absurd. □

To illustrate why this is good news, consider the case that justified the necessity of path merging in RSE: Figure 4.4, page 35. The program P has two paths π and π' starting at location s, selected depending on an uncontrolled boolean input x, and which join again in location ℓ. Neither π_1 nor π_2 satisfies single path robust reachability, but ℓ is robustly reachable. Robust reachability can "reappear" from non-robust paths quite unpredictably, so we are forced to merge all paths to keep completeness. This is not the case with quantitative reachability as Proposition 13 guarantee that one of π_1 or π_2 has quantitative reachability at least $\frac{1}{2}$:

$$q\left(P|^{\pi_i}, \ell\right) \geq \frac{1}{2} q\left(P|^{\pi_1,\pi_2}, \ell\right) = \frac{1}{2} q\left(P, \ell\right) = \frac{1}{2}$$

In this situation one can thus still detect ℓ without path merging by lowering our detection threshold by one half.

5.4.3 Comparison to other quantitative formalisms

Several domains have attempted to reason more precisely about systems by adopting a quantitative approach.

Probabilistic reachability Program verification is usually encoded as the reachability of an undesirable condition, so it is natural to consider the probability of reaching it. For example probabilistic symbolic execution [59] attempts to compute the probability[3] of each path, and

[3] Actually, they compute model counts and therefore assume uniformly distributed inputs, like we do.

shows experimentally that one can find bugs by focusing human analysis on improbable paths, presumably because those paths are badly tested by conventional techniques. The main difference with quantitative robustness is that such techniques do not consider the presence of an attacker. They compute the probability of a bug happening in a neutral environment, whereas we attempt to determine how often a bug can be triggered by the optimal attacker.

For the same reason, it does not look like probabilistic logics developed for model checking like pCTL [70] can be used for our purpose: the semantics of a pCTL formula is determined by the probability of the possible state transitions of the system, therefore one would have to encode the optimal attacker inside this transition system to obtain a meaningful result.

Quantitative information flow Quantitative information flow attempts to quantify the amount of information that an attacker can deduce from the observable (public) behavior, state or output of a system, interpreted as leakage of information. The threat model is as follows: the attacker chooses public input to a system, the defenders chose secret inputs, and the attacker attempts to deduce the secret from the public output. A central notion to achieve that is the capacity of the leakage channel: the logarithm of the number of public outputs which are possible. More formally, we count the public outputs z such that there exists a pair of (public, private) inputs leading to z. This is a distinct problem from ours, and leads to different counting problems: *projected model counting* [6] instead of f**-E-MAJSAT**.

5.5 Quantitative robust symbolic execution

Our goal here is to design a method to enumerate all locations with quantitative robustness above a threshold Q, and to know their quantitative robustness, for example to sort them from most robustly reachable to least robustly reachable.

5.5.1 Going quantitative from RSE

It is possible to adapt RSE (Section 4.5.2) for this purpose. RSE works by enumerating paths π of the program, converting them to a path constraint $\mathrm{pc}_\pi^O(a,x)$ expressing what input (a,x) make the program go along path π and reach the goal O, and check whether this formula passes the universal satisfiability test: $\exists a.\, \forall x.\ \mathrm{pc}_\pi^O(a,x)$. To obtain completeness, one can merge several paths $\pi_1, \ldots, \pi_n$ together by disjoining the corresponding path constraints in a universal satisfiability test: $\exists a.\, \forall x.\ \bigvee_{i=1}^n \mathrm{pc}_{\pi_n}^O(a,x)$.

One can make this algorithm quantitative by replacing the universal satisfiability test by a new test expressing that many inputs x make pc true for the best value of a. For simplicity, we consider a bitblasted version of the path constraint (see Section 3.2.4): pc is now a boolean formula in CNF, and inputs are now represented as boolean variables: $a \triangleq (a_1, \ldots, a_n)$ and $x \triangleq (x_1, \ldots, x_m)$. As the original domains for inputs are not necessarily a power of 2 in size, we add two boolean formulas $h_a(a)$ and $h_x(x)$ which express which combinations of boolean variables correspond to valid inputs: $\sharp(h_a) = |\mathcal{A}|$ and $\sharp(h_x) = |\mathcal{X}|$. h_a and h_x can also be used to express the effect of `assume` statements in the analyzed program.

Then we have:

$$q(P|^\pi, O) = \frac{1}{\sharp(h_x)} \max_a \sharp\left(\left(h_a(a) \wedge h_x(x) \wedge \mathrm{pc}_\pi^O(a,x)\right)\big|_a\right) \tag{5.3}$$

where $P|^\pi$ is the restriction of the program to execution path π. Note that formulas are assumed to have all their variables present, see our note about smoothness above.

By replacing universal satisfiability tests by tests that $q\left(P|^{\pi}, O\right)$ is greater than the threshold Q, we can enumerate paths which reach the goal with quantitative robustness above Q, and print $q\left(P|^{\pi}, O\right)$ for the user. We call this technique Quantitative Robust Symbolic Execution (QRSE). More specifically, operating this substitution on RSE yields QRSE (Algorithm 5) and on RSE+ (RSE plus path merging) it yields QRSE+ (QRSE plus path merging, Algorithm 6).

This allows reducing our quantitative verification problem to f-**E-MAJSAT** (Definition 19). Equation (5.3) also features a model counting term ($\sharp(h_x)$) in addition to the f-**E-MAJSAT** instance, but f-**E-MAJSAT** is harder than model counting and h_x is a smaller formula so in practice this part of the formula is negligible in terms of computation time.

Data : bound k, target O, threshold Q

$\phi := \bot$
for *path* π *in* `GetPaths` (k) **do**
  $\phi := \texttt{GetPredicate}(\pi, O)$
  $\chi := \frac{1}{\sharp(h_x)} \max_a \sharp\left(\left(h_a \wedge h_x \wedge \phi\right)|_a\right)$
  if $\chi \geq Q$ **then**
    `/* O has quantitative robustness` $\geq \chi$ `*/`
    return (true, χ)
end
return false

Algorithm 5 : QRSE: Quantitative Robust SE

Data : bound k, target O, threshold Q

$\phi := \bot$
for *path* π *in* `GetPaths` (k) **do**
  $\phi := \phi \vee \texttt{GetPredicate}(\pi, O)$
  $\chi := \frac{1}{\sharp(h_x)} \max_a \sharp\left(\left(h_a \wedge h_x \wedge \phi\right)|_a\right)$
  if $\chi \geq Q$ **then**
    `/* O has quantitative robustness` $\geq \chi$ `*/`
    return (true, χ)
end
return false

Algorithm 6 : QRSE+: Quantitative Robust SE with systematic path merging

Proposition 14 (Correctness of QRSE)**.** *If QRSE reports a target O with quantitative robustness χ, then $q\left(P, O\right) \geq \chi$.*

Proof. QRSE reaching O proves that there is a path π such that $q\left(P|^{\pi}, O\right) = \chi$. By Proposition 11, $q\left(P, O\right) \geq \chi$. □

Proposition 15 (k-completeness of QRSE+)**.** *We remind the reader that we suppose that the domain of inputs is finite. $P|^{\leq k}$ denotes the restriction of program P to traces of length at most k. Let Q be a threshold. Assuming solver termination, if a target O has quantitative robustness $q\left(P|^{\leq k}, O\right) \geq Q$, then it is reported by QRSE+ with a quantitative robustness between Q and $q\left(P|^{\leq k}, O\right)$.*

Proof. In $P|^{\leq k}$, for each possible input, there is at most one maximal path of length at most k (and all its prefixes). When QRSE+ has explored all paths, the path constraint will be equivalent to reaching O. f-**E-MAJSAT** on this path constraint will therefore have the desired value $q\left(P|^{\leq k}, O\right)$. If some subset of these paths has quantitative robustness between Q and $q\left(P|^{\leq k}, O\right)$, QRSE+ may return early. □

Note that we present QRSE as relying on functional **E-MAJSAT** and then comparing the result to the threshold Q. This could be reformulated as the original decision problem called

E-MAJSAT. We leave this approach unexplored. Obtaining the actual value instead of only the truth value of the comparison is interesting for further inspection by a human or to sort the bugs found by descending quantitative robustness, and it makes formalism slightly more intuitive. In practice, the most effective f**-E-MAJSAT** solving technique we will use is not able to take advantage of knowing the comparison threshold Q beforehand, so this will make little difference.

5.5.2 Path merging

Recall that in the qualitative case, we lose k-completeness if we do not perform path merging in robust symbolic execution. We want to avoid path merging for two main reasons: firstly, some paths can be hard to execute symbolically (*e.g.* because they contain exotic system calls, or dynamic jumps, *etc.*), and secondly, merged path constraints are more complex and harder to solve (see Section 4.6.3).

In the quantitative case, we can show that QRSE without path merging is actually as complete as QRSE with path merging under reasonable assumptions. We will use the property that if the target has high quantitative robustness and is reached by not too many paths, then one of the reaching paths must also have high quantitative robustness. We proved this form of conservation of quantitative robustness as Proposition 13. It is specific to this quantitative approach: because it is a boolean, robust reachability can vanish at a branch, and reappear later if paths rejoin.

The hypothesis "and is reached by not too many paths" may look strange at first, but it can be derived from a quite reasonable assumption:

Definition 21 (Badly scaling path merging assumption)**.** We assume that merged paths constraints are more difficult to solve than their constituents, and that there is an integer κ such that, when merging the paths constraints of more than κ paths together, the resulting path constraint is so large and/or complex that our solver will return UNKNOWN.

Under the badly scaling path merging assumption, we can prove that QRSE can be made as complete as QRSE+:

Proposition 16 (QRSE vs QRSE+)**.** *Under the badly scaling path merging assumption, all locations reported by QRSE+ as having quantitative robustness above the threshold Q are also reported by quantitative robustness above the threshold Q/κ.*

Proof. Let O be a target reported by QRSE+ with threshold Q. By the badly scaling path merging assumption, there are paths $\pi_1, \ldots, \pi_n$ with $n \leq \kappa$ such that our f**-E-MAJSAT** solver can compute $\chi \triangleq q\left(P|^{\pi_1,\ldots,\pi_n}, O\right)$ with $\chi \geq Q$. By Proposition 13, there is a path π_i such that $q\left(P|^{\pi_i}, O\right) \geq Q/n \geq Q/\kappa$. As we assume that merged path constraints are harder to solve than the original ones, our solver can compute the f**-E-MAJSAT** problem associated to $q\left(P|^{\pi_i}, O\right)$ and QRSE will thus detect O by path π_i with the threshold Q/κ. □

In practice, this means that if path merging turns out to be a problem for QRSE+ with threshold Q, then one can run QRSE with threshold Q/κ and have the guarantee of finding

- all targets with quantitative robustness above Q;
- no targets with quantitative robustness below Q/κ.

The second point ensures we keep a good signal-to-noise ratio. This principle will be illustrated in our second case study about libvncserver (Section 5.9.3.3).

5.5.3 Path pruning

We now turn to path pruning. For robust symbolic execution, we observed that we could drop paths which do not satisfy the universal satisfiability test $\exists a. \forall x.\ \mathrm{pc}_\pi$, *i.e.* paths which are not robust. This form of symbolic execution was dubbed $\mathrm{RSE}_\forall$. However, this costs some amount of completeness because such paths can sometimes be merged together to become robust again, and thus interesting. For this reason we went great length in Section 4.5.4 to recover some amount of completeness by an arguably complex form of path merging dubbed $\mathrm{RSE}_\forall$+. We can do better and simpler with QRSE.

Definition 22 (QRSE with path pruning)**.** Path pruning for QRSE is defined as follows: if during QRSE a path π is such that $q\,(P|^{\pi}, O) \leq \varepsilon$, then this path can be optionally discarded.

ε should be small, for example of the order of 10^{-9}. We expect this is feasible: remember that `prog1` in our motivating example meets this requirement.

Proposition 17 (Path pruning can remain complete)**.** *Let O be a target reachable with quantitative robustness Q. We consider QRSE+ with path pruning where path pruning is allowed to prune at most N paths. Then O is also detected with quantitative robustness at least $Q - N\varepsilon$ (assuming solver termination).*

Proof. There exists a set of M paths such that $q\,(P|^{\pi_1,\ldots,\pi_M}, O) = Q$. From those paths, at most N have been pruned: $\pi_i, i \in S$. They verify $q\,(P|^{\pi_i}, O) \leq \varepsilon$. At some point QRSE+ will consider the merger of paths $\pi_j, j \in [1, M] \setminus S$. The corresponding quantitative robustness is $\chi \triangleq q\left(P|^{\pi_j, j\in[1,M]\setminus S}, O\right)$. By proposition 12,

$$q\,(P|^{\pi_1,\ldots,\pi_M}, O) \leq \chi + \sum_{i\in S} q\,(P|^{\pi_i}, O)$$

which yields

$$Q \leq \chi + N\varepsilon$$

□

For ε of the order of 10^{-9} it means that we can easily prune millions of paths before having to lower the detection threshold too much.

In practice, this trick has not proved useful for performance for the test suite of the previous chapter (Section 4.6.3), because the number of pruned paths is low.

5.6 Formalisms for f-E-MAJSAT

We now focus on the problem of solving f-**E-MAJSAT**. f-**E-MAJSAT** has been approached independently under many forms and many names by different communities. We will present formalisms for Bayesian networks, stochastic satisfiability and model counting, and attempt to hint at the links between them.

5.6.1 Model counting

We already presented **SAT** (NP-complete) and ♯**SAT** (♯P-complete) in Chapter 3. Complexity results for function problems like ♯**SAT** are usually also given as the complexity of the corresponding decision problem: the problem where the value to be computed is above a threshold. In the case of ♯**SAT**, the corresponding decision problem is **MAJSAT**:

Definition 23. MAJSAT is the following decision problem:

Input a CNF formula f

Output whether a majority of complete assignments satisfy f, or said differently, whether $\sharp(f) \geq 2^{|V(f)|-1}$.

MAJSAT is a classic PP-complete problem. The intuitive reason is that verifying that an assignment satisfies f is in P. PP is the class of problems such that a polynomial non-deterministic Turing machine answers "true" in a majority of its non-deterministic executions. A non-deterministic machine can thus non-deterministically decide the value of each variable, then verify that the non-deterministic assignment is a model: the majority of answers is the truth value of the **MAJSAT** problem.

$\sharp$**SAT** has been extended in several directions.

5.6.1.1 Maximization

Littman, Goldsmith, and Mundhenk [87] extend **MAJSAT** into a harder one: **E-MAJSAT** ("exists" **MAJSAT**).

Definition 24. E-MAJSAT [87] is the following decision problem:

Inputs A CNF formula f and a partition of its variables: $V(f) = A \uplus X$.

Output Whether there exists an assignment $a \in \mathbb{B}^A$ of variables in A such that a majority of assignment of variables of X satisfy $f|_a$.

Variables in A are sometimes called *choice variables*, while variables in X are called *chance variables*. By reference to our intended use case, we also sometimes refer to variables in A as controlled variables and variables in X as uncontrolled variables.

E-MAJSAT is $\mathrm{NP}^{\mathrm{PP}}$-complete [87], meaning that it would become NP with a PP oracle. This is interpreted as the fact that it combines a constraint solving part (NP) with a **MAJSAT** counting part (PP).

The problem f-**E-MAJSAT** that we presented earlier (Definition 19) is the corresponding functional problem, where we remove the threshold and instead ask for the exact value formerly compared to the threshold.

5.6.1.2 Projection

E-MAJSAT extends $\sharp$**SAT** by adding a maximum operator, another direction of extension is projecting the problem on a subset of variables.

Definition 25 (projected model counting [6])**.** $\sharp\exists$SAT is the following function problem: given a propositional formula f in CNF, a subset $A \subseteq V(f)$, output $\left|\{a \in \mathbb{B}^A \mid M(a(f)) \neq \varnothing\}\right|$.

In other words, models that differ only on variables not in A are not distinguished. This problem arises naturally when one introduces auxiliary variables when modelling a system, which could influence the model count. One then uses projected model counting on the variables which were originally in the system. This happens in quantitative information flow problems, where one counts the number of possible messages of a public channel whatever the value of the secret part of state. Projected model counting is $\sharp$NP-complete [51].

We mention projected model counting not because it can be used to encode f-**E-MAJSAT**, but because some projected model counters use similar techniques to those we will present later.

5.6.1.3 Weights

Counting models of a formula is related to the probability of satisfying it with a uniformly sampled assignment. To go beyond uniform distributions one can do *weighted model counting*: each variable v is assigned a rational weight $\omega(v) \in [0,1]$, which gets mapped to a weight on literals: $\omega(\neg v) = 1 - \omega(v)$ and to complete assignments seen as a set of non-contradicting literals: $\omega(m) = \prod_{l \in m} \omega(l)$.

Definition 26. The weighted model counting problem **WMC** is the following function problem

Inputs a CNF formula f, and a weight distribution $\omega \in (\mathbb{Q} \cap [0,1])^{V(f)}$.
Output $\sum_{m \in M(f)} \omega(m)$

This problem is also ♯P-complete (it is at least as hard as ♯**SAT** and **PR** that we will introduce later reduces to it) but it represents sampling models where variables follow independent Bernoulli distributions of parameter $\omega(v)$. On an operative note, an algorithm to reduce a weighted model counting problem with dyadic weights to a standard model counting problem is described in Lee and Jiang [84]. The problem size increases linearly in the maximal 2-adic valuation of the weights. As this algorithm is meant for approximate model counting, this means that the cost is linear in the approximation quality of the original weights.

5.6.2 Stochastic Boolean satisfiability

A Stochastic boolean Satisfiability (SSAT) formula is a formula of the form

$$f = Q_1 x_1 \dots Q_n x_n \varphi(x_1, \dots, x_n)$$

where φ is a boolean formula in CNF and the Q_i are quantifiers: either the usual existential quantifier $\exists$ or the randomized quantifier with rational probability p_i, denoted as ϐ^{p_i}.

The probability of satisfying a SSAT formula is defined by:

- $\Pr(\top) = 1$;
- $\Pr(\bot) = 0$;
- $\Pr(\exists x.\, f) = \max(\Pr(f|_x), \Pr(f|_{\neg x}))$;
- $\Pr(\text{ϐ}^p x.\, f) = p \Pr(f|_x) + (1-p) \Pr(f|_{\neg x})$.

Definition 27 ([99])**. SSAT** is the following decision problem:

Input a SSAT formula f;
Output $\Pr(f) > \frac{1}{2}$.

This problem is PSPACE-complete, just like **QBF** (satisfiability of boolean formulas with arbitrary universal and existential quantifiers). One can define the corresponding functional problem as computing $\Pr(f)$.

This formalism is closely related to model counting. Model counting problems can reduce to **SSAT** problems: $\sharp(f(x_1, \dots, x_x)) = 2^n \Pr\left(\text{ϐ}^{\frac{1}{2}} x_1. \dots \text{ϐ}^{\frac{1}{2}} x_n.\, f\right)$. f**-E-MAJSAT** can be expressed as $2^{|X|} \Pr\left(\exists a.\, \text{ϐ}^{\frac{1}{2}} x.\, f\right)$. The most notable difference is that SSAT allows specifying variables with any independent Bernoulli distributions, whereas unweighted model counting is limited to equiprobable independent bits, *i.e.* the uniform distribution.

5.6.3 Bayesian networks

A Bayesian network is a set of discrete random variables over a finite domain such that:

- For each random variable X_i, there is a subset of the rest of the variables $X_{k_{i,1}}, \ldots, X_{k_{i,j_i}}$ such that the conditional probability distribution of X_i given any observation of the $X_{k_{i,l}}$ is given;
- The graph where nodes are variables and there is an edge from $X_{k_{i,l}}$ to X_i is acyclic.

(For the purpose of complexity theory results, which are taken from Park and Darwiche [100], probabilities must be given as rational numbers.)

The probability of an assignment $x = (x_1, \ldots, x_n)$ of variables is then defined from the conditional probability of each node:

$$\Pr(X = x) = \prod_{i=1}^{n} \Pr\left(X_i = x_i \mid (X_{k_{i,1}}, \ldots, X_{k_{i,j_i}}) = (x_{k_{i,1}}, \ldots, x_{k_{i,j_i}})\right)$$

A Bayesian network is a way to express the probability distributions of a set of dependent random variables by a set of constraints. It can thus express some (but not all) distributions not expressible with SSAT or model counting. In the case of n boolean variables, a Bayesian network can be much more concise than enumerating the probabilities of the 2^n possible outcomes in a naive probability table.

The cost to pay for this concision is that the probability distribution is implicit, and the problem of making parts of it explicit, known as inference, has a cost.

Definition 28. PR is the following function problem:

Input A Bayesian network over variables $X_1, \ldots, X_n$ and an evidence $x_1, \ldots, x_n$ (possible value) for variables $X_1, \ldots, X_n$.

Output $\Pr((X_1, \ldots, X_n) = (x_1, \ldots, x_n))$

PR is ♯P-complete [108]. This is the same complexity as (weighted) model counting, because the problem can be thought as counting the satisfying outcomes with their probability as weights. This analogy is more than a mere likelihood: it is possible to encode a Bayesian network as a CNF formula [27] and then defer to an existing weighted model counting algorithm [26], close to the d-DNNF-based algorithm model counting we presented earlier (Theorem 1).

The corresponding decision problem uses a threshold:

Definition 29. D-PR is the following function problem:

Input A threshold p, a Bayesian network over variables $X_1, \ldots, X_n$ and an evidence $x_1, \ldots, x_n$ (possible value) for variables $X_1, \ldots, X_n$.

Output $\Pr((X_1, \ldots, X_n) = (x_1, \ldots, x_n)) > p$

D-PR is PP-complete [89], as is expected coming from a ♯P functional problem.

More central is the problem of the *most probable explanation*. One has collected partial evidence e, *i.e.* measured the value of some variables, and one wants to deduce the most probable state x of the full network.

Definition 30 (Most probable explanation)**. MPE** is the following function problem:

Inputs A Bayesian network on variables X, and an evidence e on a subset of variables E.

Output An assignment x for all variables in X such that $\Pr(X = x \mid E = e)$ is maximal.

The corresponding decision problem **D-MPE** where one looks for an assignment with probability above a threshold is NP-complete [113].

Finally, let us mention the even more general problem called *maximum a posteriori hypothesis*, where one desires to find the most probable assignment to only a subset A of variables. Other variables are marginalized, *i.e.* their probabilities must be summed together.

Definition 31 (Maximum a posteriori hypothesis)**. MAP** is the following function problem:

Inputs A Bayesian network on variables X, an evidence e on a subset of variables E and a set of "MAP variables" A.

Output An assignment a of variables in A such that $\Pr(A = a \mid E = e)$ is maximal.

This is very close to f**-E-MAJSAT**: computing a probability is comparable to computing a (weighted) model count, and thus finding an assignment to some variables that maximizes a probability is close to finding an assignment to some variables that maximizes a model count. The corresponding decision problem **D-MAP** (does there exist a such that the probability $\Pr(A = a \mid E = e)$ is greater than a threshold p) is $\mathrm{NP}^{\mathrm{PP}}$-complete, by reduction of **E-MAJSAT** to **D-MAP** [100].

Note that the terminology around Bayesian networks features some diversity: Bayesian networks are sometimes called belief propagation networks [113]; **MPE** is also known as **MEU** (maximum expected utility) or, confusingly, **MAP** (maximum a posteriori hypothesis) [113], in which case what we called **MAP** above is denoted as **MMAP** (marginal MAP) [122].

5.6.4 Probabilistic Planning

Finally, let us present a family of problems called probabilistic planning. They elicited some of the developments and algorithms we will present in the next section.

A probabilistic planning problem (or *domain*) is a combination of a finite set S of possible world states, a distribution $\mathcal{I}$ describing the (uncertainty) of the initial state, a finite set of possible actions A, a set of goal states $G \subseteq S$, and a *horizon* $n \in \mathbb{N}$. An actor must plan n actions in the following sense: n times, it must measure some (but not all) of the variables of the state, possibly with added uncertainty, and take an action in A, which may have a probabilistic effect on the state of the system. The goal of the actor is to *plan* its actions to maximize the probability of reaching a state in G after n actions.

A standard example of domain is the Slippery-Gripper domain. A robot must paint a block by holding it in its gripper. The robot has three possible actions: pick the block up, dry its gripper, and paint the block. Each action has a probability of failing, *i.e.* not have the intended effect. If the gripper is wet, the probability of failing to pick the block up is increased, so there is an interest in using the "dry the gripper" action. The action "paint" may dirty the gripper, and must not be performed blindly as block must not be painted twice, *etc.* The horizon n expresses how many actions the robot can use to attempt to paint the block. It can use sensors to obtain (uncertain) information on the state of the system: whether the gripper is clean, wet, *etc.*

The existence of a plan with probability above a threshold is EXPTIME-complete in the general case, but when the horizon n grows polynomially it becomes PSPACE-complete because of a strong likeness to SSAT[88].

Informally, it is possible to describe the k-th action as a set of boolean variables a_k, the state after k actions as a set of boolean variables s_k and the added uncertainty on measures and actions as some boolean random variables r_k. There is a formula ϕ expressing the relation

between the former state and the measurements, and between the action and the next state such that reaching the goal can be expressed as a SSAT problem:

$$\forall r_0 s_0.\, \exists a_1.\, \forall r_1 s_1.\, \exists a_2.\, \ldots \forall r_{n-1} s_{n-1}.\, \exists a_n.\, \forall r_n s_n.\, I(r_0, s_0) \wedge \bigwedge_{i=1}^{n} \phi(s_{i-1}, a_i, r_i, s_i) \wedge g(s_n)$$

where I is for the initial state and g expresses that the final state is a goal state [75].

In the case of *conformant* probabilistic planning, *i.e.* the special case where the actor has no sensors, then all actions must be predetermined and the corresponding SSAT problem looks like $\exists \text{actions}.\, \forall \text{chance}.\, \text{goal reached}$ which is the SSAT pendant to **E-MAJSAT**.

5.6.5 Summary

We presented 3 types of problems (summarized in Table 5.1): problems generalizing model counting where all models originally have the same weight, problems generalizing boolean satisfiability with independent Bernoulli distributions on individual bits and arbitrary quantification (**SSAT**), and problems where the distribution of variables is given implicitly as a Bayesian network. Model counting problems can be seen as special cases of **SSAT** with few quantifiers and uniform weights; conversely the need for non-uniform weights or probabilities led the model counting community to consider weighted model counting, which can encode more of **SSAT** than mere model counting; Bayesian network inference problems can encode distributions that **SSAT** cannot but there are techniques to encode Bayesian networks into **SSAT** more indirectly. In the end, most of these problems can be encoded as model counting, weighted model counting, or a variant, which explains why research on Bayesian network ended up benefitting model counting research.

These problems fall into several categories: computing a model count or probability, to describe a system; inference, where one want to compute the most likely state of a system from partial observation; and planning where one wants to compute the best course of events to obtain a desirable outcome. Bayesian networks, being an implicit statistical description of a complex system, focus mostly on inference, while communities handling fully describable systems tend to use the **SSAT** probabilistic planning formulation, but it turns out that in the case where only one action is to be performed, they are similar problems: a weighted form or f-**E-MAJSAT**.

Our problem of computing quantitative robustness can be seen as a dumbed-down conformant probabilistic planning problem, hence the long digression of this section. It has one notable difference, which will come into play when we talk about tools: we will consider many instances where no plan has a high success probability (non-vulnerable paths), whereas both in the case of inference and planning, existing research focused on finding good plans.

5.7 Algorithms for f-E-MAJSAT

In this section we describe how to solve f-**E-MAJSAT**, or close problems like **MAP** and two-quantifiers **SSAT**. There are a number of such solvers in the literature, and given that f-**E-MAJSAT** is a hard problem, each of them is possibly tailored to the kind of problems that first elicited its development. We thus consider various algorithms, and will compare many of them experimentally in Section 5.9.2.

5.7.1 DC-SSAT

DC-SSAT [92] is a solver for SSAT problems coming from Completely Observable Probabilistic Planning. These SSAT problems have arbitrarily many quantifier alternations starting with an

Table 5.1: Summary of the functional problems and corresponding decision problems we presented, along with some well-known ones for context

	Formulation	Function problem	Complexity	Expression in **SSAT**	Decision Problem	Complexity	Ex. domains of application
	$\exists y.\, f(y)$			$\exists y.\, f(y)$	**SAT**	NP-complete	Constraint solving, model checking
Model counting	$\sharp\,(f)$	$\sharp$**SAT**	$\sharp$P-complete	$\mathbb{R}^{\frac{1}{2}} y.\, f(y)$	**MAJSAT**	PP-complete	Statistical description of systems
	$\max_a \sharp\,(f|_a)$	f**-E-MAJSAT**		$\exists a.\, \mathbb{R}^{\frac{1}{2}} x.\, f(a,x)$	**E-MAJSAT**	$\mathrm{NP}^{\mathrm{PP}}$-complete	Probabilistic inference, probabilistic planning
	$\lvert\{a \mid \sharp\,(f|_a) \neq 0\}\rvert$	$\exists\sharp$**SAT**	$\sharp$NP-complete	$\mathbb{R}^{\frac{1}{2}} a.\, \exists x.\, f(a,x)$			Quantitative information flow
	$\sum_{y:f(y)} \prod_i \omega(y_i)$	**WMC**	$\sharp$P-complete	$\mathbb{R}^{p_i} y_i.\, f(y)$			
	probability of satisfaction, independent bits $Q_i y_i.f(y)$, $Q_i \in \{\exists, \forall\}$	**SSAT**		$Q_i^{p_i} y_i.\, f(y)$, $Q_i \in \{\exists, \mathbb{R}\}$	**SSAT**	PSPACE-complete	Probabilistic planning
					QBF	PSPACE-complete	Planning, model checking
Bayesian net.	probability in Bayesian networks	**PR**	$\sharp$P-complete		**D-PR**	PP-complete	Statistics
	$\operatorname{argmax}_X \Pr(X|e)$, X complete	**MPE**			**D-MPE**	NP-complete	Fault diagnosis, explanation
	$\operatorname{argmax}_A \Pr(A|e)$, A partial	**MAP**			**D-MAP**	$\mathrm{NP}^{\mathrm{PP}}$-complete	Fault diagnosis, explanation

existential one $\exists x_1.\, \phi_1 \wedge (\text{ꓤ} x_2.\, \phi_2 \cdots \wedge (\text{ꓤ} x_n.\, \phi_n) \dots)$ and ϕ_k can only contain variables in adjacent quantifiers: x_{k-1}, x_k and x_{k+1}.

DC stands for divide and conquer: DC-SSAT first enumerates solutions (called viable partial assignments, VPA) for the ϕ_k individually, taking into account that solutions generated for adjacent quantifier blocks must agree on existentially quantified variables. Dividing the original problem into such smaller sub-problems allows the use of aggressive caching (or memoization), especially because the natural encoding of Probabilistic Planning problems into SSAT leads to the ϕ_k having a very similar structure. DC-SSAT then explores the tree of VPAs in the search of a leaf (complete assignment) with maximal probability.

As a summary, DC-SSAT is designed for SSAT problems with specific structure and arbitrary quantifier alternations in mind, and later benchmarks [85] suggest even after the years it still shows a significant performance advantage for this kind of problems.

5.7.2 Maxcount

MAXCOUNT [57] attempts to solve the model counting equivalent of computing the probability of $\exists a.\, \text{ꓤ}^{\frac{1}{2}} b.\, \exists c.\, \varphi$: finding the assignment to a such that $\exists c.\ \varphi|_a$ is true for a maximum number of values of b. This problem is dubbed **Max♯SAT**, and f-**E-MAJSAT** can obviously be expressed in terms of this **Max♯SAT**.

MAXCOUNT relies on the following observation: when $\exists c_1, \dots, c_k.\ \bigwedge_{i=1}^{k} \varphi(a, b_i, c_i)$ has n models for a and $b_1, \dots, b_k$, the solution to **Max♯SAT** should be close to $\sqrt[k]{n}$. For k chosen of the order of the number of boolean variables in a, Fremont, Rabe, and Seshia provide explicit bounds on the quality of the result: the actual solution is ε-close to the computed answer with $1 - \delta$ probability, for ε and δ arbitrary small parameters.

In practice, the formula on which model counting is performed is of size linear in k, which makes model counting orders of magnitude harder than on the original φ, so k is experimentally chosen small (1 to 13), yielding correct probabilistic bounds but which may be less precise than for k chosen as said above.

Experimental evaluation of MAXCOUNT was mostly directed towards quantitative information flow and program synthesis.

5.7.3 ssatABC

SSATABC [85] attempts to solve SSAT problems of the form $\exists a.\, \text{ꓤ} x.\, \varphi$, called **ER-SSAT** for exists-random SSAT, and to which f-**E-MAJSAT** maps easily. It imports the technique called clause selection from Quantified Boolean Formula (QBF) solvers. For a given a, one observes the set of clauses $S(a)$ in $\varphi|_a$ that are left unsatisfied by a. Such clauses are said to be *selected*. For any a' that selects more clauses than a ($S(a) \subseteq S(a')$), we have $\Pr(\text{ꓤ} x.\ \varphi|_{a'}) \leq \Pr(\text{ꓤ} x.\ \varphi|_a)$ because, informally, a' constrains φ more than a and leaves fewer models. We can deduce that no such a' can be a solution to the problem, and we want to retain this information. The idea of clause selection is to add variables that are equivalent to "clause k is satisfied", which allows learning from a a clause expressing "at least one of the clauses in $\varphi|_a$ must be deselected, *i.e.* satisfied".

SSATABC works by incrementally sampling assignments to a that satisfy φ conjoined to the clauses learned as explained above with a SAT solver until said formula is unsatisfiable. It defers to a weighted model counter to obtain the probability of the best assignment. In practice, the model counter is also used during the refinement loop for some more advanced optimizations.

SSATABC has been experimented on many kinds of formulas: probabilistic planning encoded into SSAT, QBF instances where universal quantifiers were replaced by ꓤ quantifiers, the

test suite of MAXCOUNT [57] were extra quantifiers were similarly replaced, and formulas from maximum probabilistic equivalence checking.

5.7.4 d-DNNF-based techniques

Several techniques based on d-DNNF-compilation have been proposed. They all have in common that while they often claim relying on d-DNNF-compilation, they in fact use the stronger decision-DNNF form, which can be obtained without loss of generality.

In the following discussion, we denote the result of f**-E-MAJSAT** on f with choice variables in A and chance variables in X as:

$$\mathrm{emajsat}_A(f) \triangleq \max_{a \in \mathbb{B}^A} \sharp(f|_a)$$

As said in Theorem 1 once a CNF formula is converted to decision-DNNF, one can obtain its model count by considering And nodes as multiplication and Ite nodes as addition.

One can extend this result to f**-E-MAJSAT**, by adding a further constraint on the form of the decision-DNNF:

Definition 32. A formula in decision-DNNF is said to be $(A_1, \ldots, A_n)$-layered if $V(f) \subseteq \biguplus_{i=1}^{n} A_i$ (where $\uplus$ denotes disjoint union) and for any ite node $\mathrm{ite}(v, f, g)$, we have $v \in A_i \implies V(f) \cup V(g) \subseteq \bigcup_{j=i}^{n} A_j$.

This corresponds to Ite nodes on variables in A_1 on top, then those on A_2 below and so on. Some decision-DNNF compilers like Dsharp [94] can produce layered decision-DNNF because it can be used for projected model counting [81]: it is enough to put projection variables in the upper layer and to modify Theorem 1 to consider that all subtrees in the lower layer variables have model count 1.

We can now solve f**-E-MAJSAT** on layered decision-DNNF (example in Figure 5.3):

Proposition 18 (Constrained algorithm)**.** f***-E-MAJSAT*** *over $A \subseteq \mathcal{V}$ of a formula f in $(A, \mathcal{V} \setminus A)$-layered smooth decision-DNNF can be computed by:*

$$\mathrm{emajsat}_A(\top) = 1 \tag{5.4}$$

$$\mathrm{emajsat}_A(\bot) = 0 \tag{5.5}$$

$$\mathrm{emajsat}_A(\mathrm{ite}(v, g, h))) = \mathrm{emajsat}_A(g) + \mathrm{emajsat}_A(h) \qquad \text{when } v \notin A \tag{5.6}$$

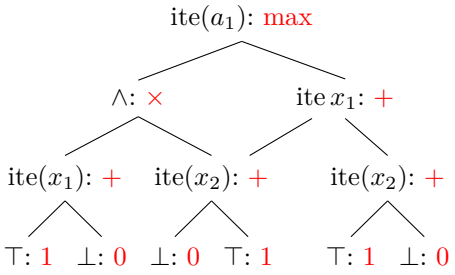
$$\mathrm{emajsat}_A(\mathrm{ite}(v, g, h))) = \max(\mathrm{emajsat}_A(g), \mathrm{emajsat}_A(h)) \qquad \text{when } v \in A \tag{5.7}$$

$$\mathrm{emajsat}_A\left(\bigwedge_{i=1}^{n} g_i\right) = \prod_{i=1}^{n} \mathrm{emajsat}_A(g_i) \tag{5.8}$$

Said otherwise, And nodes map to multiplication, chance Ite nodes to addition and choice Ite nodes to maximum.

It is also possible to construct a witness partial model, *i.e.* a partial model $a \in \mathbb{B}^A$ with (maximal) model count $\mathrm{emajsat}_A(f)$.

Proposition 19. *For f a formula in $(A, \mathcal{V} \setminus A)$-layered decision-DNNF one defines $w_A(f) \in \mathbb{B}^A$*



For $f = \text{ite}(a_1, x_1 \wedge \neg x_2, \text{ite}(x_1, \neg x_2, x_2))$, Proposition 18 yields $\text{emajsat}_{\{a_1\}}(f) = \max((1+0) \times (0+1), (0+1) + (1+0)) = 2$.

Figure 5.3: A formula in decision-DNNF (black), with f-**E-MAJSAT** solving for it (red).

inductively as follows:

$$w_A(g) = a_\perp \quad \textit{if } V(g) \subseteq \mathcal{V} \setminus A$$

$$w_A(\text{ite}(v, g, h)) = \begin{cases} w_A(h)[v := \perp] & \textit{if } \text{emajsat}_A(f) = \text{emajsat}_A(h) \\ w_A(g)[v := \top] & \textit{otherwise} \end{cases} \textit{ for } v \in A$$

$$w_A\left(\bigwedge_{i=1}^{n} g_i\right) = g_1 || \ldots || g_n$$

where $a_\perp$ denotes the partial valuation where all variables in A are mapped to $\perp$, and $a[v := x]$ denotes the valuation that maps v' to x if $v = v'$ else to $a(v')$.

Then one has: $\sharp\left(f|_{w_A(f)}\right) = \text{emajsat}_A(f)$.

Note that the 3 cases cover all possibilities by layering hypothesis. For and nodes in the lower layer, both the first rule and the last one match, but they yield the same result.

To our knowledge this algorithm has no name in the literature, it is mentioned in Huang [75] and Pipatsrisawat and Darwiche [101] as a straightforward technique that is not practical in terms of performance and upon which they intend to improve. We will call this algorithm Constrained. The lack of performance of Constrained comes from the fact that obtaining a constrained (layered) decision-DNNF is significantly more expensive than an unconstrained one. Compilation is exponential in the minimum width of all compatible variable orders, and we constrain the compiler to only use variables orders where variables in A come before variables in X.

Finally, let us mention that Constrained can be extended to handle weights (by adding them on Ite nodes in eq. (5.6)) and **Max♯SAT** (by adding one layer: to compute $\Pr(\exists a. \text{ϐ} x. \exists z. f)$ one needs a (A, X, Z)-layered decision-DNNF formula where one considers the value of Z subtrees as 1 if they are consistent or 0 if they are not). As we focus on f-**E-MAJSAT** here, we leave these details aside for the sake of simplicity.

Unconstrained decision-DNNF One can circumvent the increased cost of a constrained decision-DNNF by satisfying one-self with an upper bound. If one applies Proposition 18 on an unconstrained (without layering constraint) formula, one obtains an upper bound:

Definition 33 (Unconstrained algorithm)**.** Let f be a decision-DNNF formula, not necessarily layered. One defines N inductively as follows:

$$N(\top) = 1 \tag{5.9}$$
$$N(\bot) = 0 \tag{5.10}$$
$$N(\text{ite}(v, g, h))) = N(g) + N(h) \qquad \text{when } v \notin A \tag{5.11}$$
$$N(\text{ite}(v, g, h))) = \max(N(g), N(h)) \qquad \text{when } v \in A \tag{5.12}$$
$$N\left(\bigwedge_{i=1}^{n} g_i\right) = \prod_{i=1}^{n} N(g_i) \tag{5.13}$$

Proposition 20. $N(f) \geq \text{emajsat}_A(f)$.

This algorithm, which we call UNCONSTRAINED, is still linear in the size of the decision-DNNF, and has the advantage of requiring a cheaper compilation step. It follows exactly the same principle as CONSTRAINED: multiplication for And nodes, addition for chance Ite nodes, and maximum for choice Ite nodes.

Complan COMPLAN [75] was designed with Conformant Probabilistic Planning problem instances translated to SSAT in mind: these correspond to SSAT formulas with one quantifier alternation $\exists a. \text{ꓤ} x. f$.

UNCONSTRAINED is at the core of COMPLAN, but COMPLAN complements it with a branch and bound algorithm to obtain an exact result for the f**-E-MAJSAT** problem at hand. Observe that from a decision-DNNF f and a partial assignment of choice variables a it is possible to obtain a decision-DNNF for $f|_a$, and therefore $N(f|_a)$ an upper bound for $\max_{a'} \sharp\left(f|_{a||a'}\right)$. Now consider a loop starting from a partial assignment a and a lower bound $m = 0$ of $\text{emajsat}_A(f)$. We attempt to explore the exhaustive search space of possible assignments to incrementally improve m until the maximal value. If for some added literal l $N(f|_{a||l})$ is lower than m, then no assignments containing a and l can lead to a high enough model count: one can learn $\neg l$ by appending it to a. Otherwise, one explores assignments corresponding to l and $\neg l$ separately.

In other words, COMPLAN explores the full search space of assignments to choice variables, pruning parts of it that are provably below the current lower bound with UNCONSTRAINED. It uses decision-DNNF compilation as a pre-computation to get fast bounds on f**-E-MAJSAT** on partial assignments of choice variables.

Complan+ COMPLAN+ [101] uses the same structure as COMPLAN to solve f**-E-MAJSAT**: an upper bound algorithm plus branch and bound to turn the upper bound into an exact result. COMPLAN+ replaces the upper bound with a more precise one, which we will designate as OVAL. Its principle is quite technical; for our purpose it suffices to say that it is always more precise than UNCONSTRAINED, and that it executes in $O(|f||A|)$ where $|f|$ is the size of the decision-DNNF and $|A|$ denotes the number choice variables.

COMPLAN+ was tested on probabilistic planning problems where it significantly improves over COMPLAN, as well as on MAP problems. The MAP counterpart to COMPLAN+ is called ACEMAP+, introduced in the same article [101]. It follows the same idea, but we do not present it here in more details.

5.7.5 Summary

We presented a number of algorithms that can solve f**-E-MAJSAT**, as summarized in Table 5.2. Some of them were repurposed: MAXCOUNT was designed to solve a form of f**-E-MAJSAT**

Table 5.2: Summary of tools and algorithms usable to solve f-**E-MAJSAT** problems

Algorithm	Target problem	**SSAT** encoding	Solution	Principle	Originally tested on
DC-SSAT	Completely Observable Probabilistic Planning encoded in **SSAT**	$(\exists\!\!\text{R})^k f$	exact	Divide and conquer on quantifier blocks	Probabilistic planning
MAXCOUNT	**Max♯SAT**	$\exists a.\, \text{R}^{\frac{1}{2}} x.\, \exists z.\, f(a,x,z)$	probabilistic interval	Self-composition	Quantitative information flow, program synthesis
SSATABC	**ER-SSAT**	$\exists a.\, \text{R} x.\, f(a,x)$	exact	Incremental improvement of a witness by clause selection	Probabilistic planning, quantitative information flow, translated QBF, …
CONSTRAINED	f-**E-MAJSAT**	$\exists a.\, \text{R}^{\frac{1}{2}} x.\, f(a,x)$	exact	◇ Constrained decision-DNNF compilation	(mentioned but untested in [75])
UNCONSTRAINED	f-**E-MAJSAT**	$\exists a.\, \text{R}^{\frac{1}{2}} x.\, f(a,x)$	upper-bound	◇ Unconstrained decision-DNNF compilation	(component of COMPLAN)
COMPLAN	f-**E-MAJSAT**	$\exists a.\, \text{R}^{\frac{1}{2}} x.\, f(a,x)$	exact	◇ Incremental improvement of a witness by branch-and-bound + UNCONSTRAINED	Probabilistic planning
OVAL	f-**E-MAJSAT**	$\exists a.\, \text{R}^{\frac{1}{2}} x.\, f(a,x)$	upper-bound	◇ Unconstrained decision-DNNF compilation	(component of COMPLAN+)
COMPLAN+	f-**E-MAJSAT**	$\exists a.\, \text{R}^{\frac{1}{2}} x.\, f(a,x)$	exact	◇ Incremental improvement of a witness by branch-and-bound + OVAL	**MAP** and probabilistic planning

Arrows denote algorithms which claim to improve over another algorithm.
◇ marks techniques based on compilation to decision-DNNF.

In the next section we propose a new algorithm:

RELAX	f-**E-MAJSAT**	$\exists a.\, \text{R}^{\frac{1}{2}} x.\, f(a,x)$	interval	◇ Partially constrained decision-DNNF compilation	QRSE

with projection (one more existential quantifier); DC-SSAT was designed for more quantifier alternations. Some methods are approximate, but in the case of UNCONSTRAINED and OVAL they were only meant to be part of an exact algorithm via a standard branch-and-bound construct, COMPLAN and COMPLAN+ respectively.

It is interesting to note that some of these tools are clearly geared toward instances with high model count: MAXCOUNT, DC-SSAT and SSATABC report the proportion of satisfying chance models as a float with fixed precision, which can underflow if this proportion is too low. For example, the median value of $\mathrm{emajsat}_A(f)$ computed in the experimental evaluation of SSATABC [85] is above 0.1. For comparison, in our motivating example (Section 5.2), `prog1` leads to a value of $\mathrm{emajsat}_A(f)$ of $4.36202 \cdot 10^{-47}$ because of the auxiliary variables of bitblasting. SSATABC represents this as 0, because it uses 32-bit floats.

As we will see in our experimental evaluation of Section 5.9.2, while these algorithms can perform well on the formulas they were designed for, they underperform when applied on formulas obtained from QRSE. For this reason we would like to propose a new technique that handles this new kind of formulas better. As SSATABC, COMPLAN+ and CONSTRAINED tend to blow up exponentially as the number of variables increases on a family bit-vector SMTLib2 formulas, while techniques based on unconstrained decision-DNNF scale significantly better, we will direct our efforts to designing an algorithm which uses a decision-DNNF that remains unconstrained to the greatest possible extent. OVAL already fits this description, but it turns out to return a coarse upper-bound. In the next section, we propose an algorithm which strikes a compromise between the precision of CONSTRAINED and the speed of OVAL.

5.8 A method to bridge the gap between constrained and unconstrained decision-DNNF based on relaxation

In this section we seek to relax the layering constraint on decision-DNNF compilation in CONSTRAINED so as to obtain a compromise between the precision of the result and the efficiency of compilation. Specifically, we ask for $(A \uplus R, X \setminus R)$-layered decision-DNNF instead of (A, X)-layered previously, with R meant to be small. This allows the compiler to do decisions on $A \cup R$ instead of just A, and reduces the treewidth of the problem.

5.8.1 Upper bound

We adapt UNCONSTRAINED (Definition 33) to obtain an upper bound on the f-**E-MAJSAT** problem.

Definition 34 (Relaxed upper bound)**.** Let f a formula in $(A \uplus R, X)$-layered smooth decision-DNNF. We define $U(f) \in \mathbb{N}$ inductively as follows:

$$U(\top) = 1 \tag{5.14}$$

$$U(\bot) = 0 \tag{5.15}$$

$$U(\mathrm{ite}(v, g, h))) = U(g) + U(h) \qquad \text{for } v \in X \tag{5.16}$$

$$U(\mathrm{ite}(v, g, h)) = \max(U(g), U(h)) \qquad \text{for } v \in A \tag{5.17}$$

$$U(\mathrm{ite}(v, g, h)) = U(g) + U(h) \qquad \text{for } v \in R \tag{5.18}$$

$$U\left(\bigwedge_{i=1}^{n} g_i\right) = \prod_{i=1}^{n} U(g_i) \tag{5.19}$$

Proposition 21. $U(f) \geq \text{emajsat}_A(f)$.

Proof. We prove the result by induction on the structure of f.

When we compute $U(g)$ for g in the lower layer of f, only eqs. (5.14) to (5.16) are used. These coincide with computation of $\text{emajsat}_{A\cup R}(g)$ in Proposition 18, but since $V(g) \cap R = \varnothing$, $U(g) = \text{emajsat}_{A\cup R}(g) = \text{emajsat}_A(g)$.

In the case of $U(f_A)$, where $f_A = \text{ite}(v, g, h)$, $v \in A$, observe that

$$\text{emajsat}_A(f_A) = \max(\text{emajsat}_A(g), \text{emajsat}_A(h))$$

By induction hypothesis, $\text{emajsat}_A(g) \leq U(g)$ and $\text{emajsat}_A(h) \leq U(h)$. As max is non-decreasing in both its arguments, we prove the desired result $\text{emajsat}_A(f_A) \leq \max(U(g), U(h))$.

Same reasoning works for the product on decomposable And nodes.

The interesting case is the case of a relaxed Ite node: $f_R = \text{ite}(v, g, h)$, where $v \in R$ (eq. (5.18)). As $v \wedge g$ and $\neg v \wedge h$ have no common model, $M(f_R) = M(v \wedge g) \uplus M(\neg v \wedge h)$. Therefore, for a partial model $a \in \mathbb{B}^A$, we have $\sharp(f_R|_a) = \sharp((v \wedge g)|_a) + \sharp((\neg v \wedge h)|_a) = \sharp(g|_a) + \sharp(h|_a) \leq \text{emajsat}_A(g) + \text{emajsat}_A(h)$. Hence, $\text{emajsat}_A(f_R) \leq \text{emajsat}_A(g) + \text{emajsat}_A(h)$. By induction hypothesis $\text{emajsat}_A(g) \leq U(g)$ and $\text{emajsat}_A(h) \leq U(h)$, and thus $\text{emajsat}_A(f_R) \leq U(h) + U(g)$. □

The principle is the same as before; what is new is that relaxed Ite nodes map to addition like chance Ite nodes, whereas during compilation they are in the upper layer like choice variables.

5.8.2 Lower bounds

The literature is mostly interested in upper bounds for f-**E-MAJSAT**, as they use it for branch-and-bound algorithms. We plan to use the upper bound as a final result, so we need a lower bound as well. We propose several of them, which we will compare experimentally in Section 5.9.2.

The first one is symmetrical to Proposition 21: for relaxed Ite nodes, we conservatively take the maximum. Then operations are the same as for Constrained (Proposition 18) when computing $\text{emajsat}_{A\cup R}(f)$:

Definition 35 (Bad lower bound)**.** Let f a formula in $(A \uplus R, X)$-layered smooth decision-DNNF. We define $L_1(f) \triangleq \text{emajsat}_{A\cup R}(f)$. We call $L_1(f)$ "bad" lower bound.

Proposition 22. $L_1(f) \leq \text{emajsat}_A(f)$.

Proof. Let $a' \in \mathbb{B}^{A\cup R}$ be a partial assignment. Let us write it as $a' = a||r$ with $a \in \mathbb{B}^A$ and $r \in \mathbb{B}^R$. If we consider assignments as set of literals, a partial assignment m is compatible with a complete one m' when $m \subseteq m'$.

$$\begin{aligned}
\sharp(f|_{a'}) &= |\{m \setminus (a \cup r) \mid m \in M(f), a \subseteq m \wedge r \subseteq m'\}| \\
&\leq |\{m \setminus a \mid m \in M(f), a \subseteq m \wedge r \subseteq m'\}| \\
&\leq |\{m \setminus a \mid m \in M(f), a \subseteq m\}| \\
&= \sharp(f|_a)
\end{aligned}$$

Therefore $\max_{a' \in \mathbb{B}^{A\cup R}} \sharp(f|_{a'}) \leq \max_{a\in\mathbb{B}^A} \sharp(f|_a)$ which is another way to write $\text{emajsat}_{A\cup R}(f) \leq \text{emajsat}_A(f)$. □

Since we compute $\text{emajsat}_f(A \cup R)$ we might as well also compute a witness $w_{A\cup R}(f)$ for it (Proposition 19 shows it is also computable in linear time): its model count is maximal for $A \cup R$ in the sense that $\sharp\left(f|_{w_{A\cup R}(f)}\right) = \text{emajsat}_f(A \cup R)$; we can expect it to have good model count when restricted to A.

Definition 36 (Fast lower bound)**.** Let f a formula in $(A \uplus R, X)$-layered smooth decision-DNNF. Let $w \in \mathbb{B}^A$ be the partial assignment coinciding with $w_{A\cup R}(f)$ on A. We define $L_2(f) = \sharp(f|_w)$ and call this lower bound "fast".

$L_2(f)$ is a lower bound by definition of $\text{emajsat}_A(f)$.

Computing $L_2(f)$ corresponds to instrumenting the computation of $L_1(f)$ by computing w as follows: on controlled Ite nodes, assign the decision variable to the boolean value of the branch that has greater L_1 value; on And notes, concatenate all w values corresponding to children. At the end compute $\sharp(f|_w)$, in linear time.

Proposition 23 (Bad lower bound is worse than fast lower bound)**.** $L_1(f) \leq L_2(f)$

Proof. In the proof of Proposition 22 we proved that when $a' = a||r$, we have $\sharp(f|_{a'}) \leq \sharp(f|_a)$. For $a' = w_{A\cup R}(f)$ and thus $a = w$ we obtain the desired result. □

Finally, we present a third lower bound which is more precise but in quadratic time. We start from the same basis: we simultaneously compute a lower bound and a witness of the lower bound. Except for relaxed Ite nodes, we keep the same principle as in Constrained (Proposition 18): multiplication of lower bound and concatenation of witness on And nodes, addition of lower bound on chance non-relaxed Ite nodes, and maximum of model count selecting the branch reaching the maximum in the witness on controlled Ite nodes. The difference is in handling relaxed Ite nodes: taking the maximum of model counts like before is only a conservative lower bound; instead of guessing we can compute the model count associated to each witnesses in linear time and select the best one.

Definition 37 (Precise lower bound)**.** Let f a formula in $(A \uplus R, X)$-layered smooth decision-DNNF. We define $W(f) \in \mathbb{B}^A$ and $L_3(f) \in \mathbb{N}$ inductively as follows:

$$(W(\top), L_3(\top)) = (a_\bot, 1) \tag{5.20}$$

$$(W(\bot), L_3(\bot)) = (a_\bot, 0) \tag{5.21}$$

$$(W(f_X), L_3(f_X)) = (a_\bot, L_3(g) + L_3(h)) \quad \text{for } f_X = \text{ite}(v, g, h), v \in X \tag{5.22}$$

$$(W(f_A)), L_3(f_A)) = \begin{cases} (W(h), L_3(h)) & \text{if } L_3(g) \leq L_3(h) \\ (W(g), L_3(g)) & \text{otherwise} \end{cases} \quad \text{for} \quad \begin{array}{l} f_A = \text{ite}(v, g, h), \\ v \in A \end{array} \tag{5.23}$$

$$(W(f_R)), L_3(f_R)) = \begin{cases} (W(h), L_3(g) + L_3(h)) \\ \quad \text{if } \sharp\left(f_R|_{W(g)}\right) \leq \sharp\left(f_R|_{W(h)}\right) \\ (W(g), L_3(g) + L_3(h)) \\ \quad \text{otherwise} \end{cases} \quad \text{for} \quad \begin{array}{l} f_R = \text{ite}(v, g, h), \\ v \in R \end{array} \tag{5.24}$$

$$(W(f_\wedge), L_3(f_\wedge)) = \left(W(g_1)||\ldots||W(g_n), \prod_{i=1}^{n} L_3(g_i)\right) \quad \text{for } f_\wedge = \bigwedge_{i=1}^{n} g_i \tag{5.25}$$

Proposition 24. $\sharp\left(f|_{W(f)}\right) = L_3(f) \leq \text{emajsat}_A(f)$.

Proof. We prove $\sharp\left(f|_{W(f)}\right) = L_3(f)$ by induction. The fact that $L_3(f)$ is a lower bound of $\text{emajsat}_A(f)$ then derives from the definition of emajsat.

$L_3(f)$ is the exact model count of f when f is a chance subtree (eqs. (5.20) to (5.22)) because these rules are identical to Proposition 18.

eqs. (5.23) and (5.24) report a lower bound equal to the witness by design.

Finally, because And nodes are decomposable the model count corresponding to a concatenation is really the product in eq. (5.25). □

This bound is computed in quadratic time instead of linear time for fast lower bound, but it can be more precise:

Proposition 25 (Precise lower bound improves over fast lower bound)**.** $L_2(f) \leq L_3(f)$

Proof. We prove this result by induction on f. As said before, rules to compute L_2 and L_3 are the same except for relaxed Ite nodes. It is therefore enough to prove that this inequality is preserved across a relaxed Ite node. In this case L_2 computes a maximum on children, while L_3 computes the actual model count of the witnesses of the children to select the better children. As the actual computation yields a higher result than the bound (Proposition 23) L_3 yields a higher result than L_2. □

5.8.3 Quality of the resulting interval

The algorithm we propose, which we call Relax, is as follows:

Definition 38 (Relax)**.** Given a formula f in CNF, a partition of its variables in $A \uplus X$, and $R \subseteq X$, first compile f to a $(A \uplus R, X \setminus R)$-layered decision-DNNF, then compute an interval $[L_2(f), U(f)]$ for $\text{emajsat}_A(f)$ with Definitions 34 and 36.

The second step is done in linear time in the size of the decision-DNNF. We will investigate the advantages of replacing fast lower bound L_2 by bad (L_1) or precise (L_3) in Section 5.9.2. Unless otherwise specified, Relax is using fast lower bound.

The main parameter of Relax is R the set of relaxed variables. R is meant to be small enough to give good approximation, but large enough to allow tractable compilation. We now turn to the influence of the size of R on size of the resulting interval.

In the limit case where R is empty (no relaxation), the algorithm becomes identical to Constrained, and the resulting interval becomes a singleton.

Proposition 26 (Relax degenerates to Constrained)**.** *If $R = \varnothing$, then $U(f)$, $L_1(f)$, $L_2(f)$ and $L_3(f)$ are all equal to* $\text{emajsat}_A(f)$.

Proof. In this case, eq. (5.18) is not used to compute U, and observe that the other rules computing U are identical to those of Proposition 18.

By definition, $L_1(f) = \text{emajsat}_{A \cup R}(f)$ but R is empty so $L_1(f) = \text{emajsat}_A(f)$.

We then deduce from Propositions 23 and 25 that L_2 and L_3 are also equal to $\text{emajsat}_A(f)$. □

Conversely, when R contains all of X, the algorithm becomes identical to Unconstrained:

Proposition 27 (Relax degenerates to Unconstrained)**.** *If $R = X$, then $U(f) = N(f)$ where N was defined in Definition 33.*

Proof. In this case, eq. (5.16) is not used to compute U, and observe that the other rules computing U are identical to those for N in Definition 33, with eq. (5.18) corresponding to eq. (5.11). □

Theorem 2 (Quality of bad bounds)**.** $U(f) \leq 2^{|R \cap V(f)|} L_1(f)$

Proof. We show the result by induction.

For base cases $\top$ and $\bot$, $U(f) = L_1(f)$.

For an Ite node with variable in X, $U(f) = L_1(f) = \sharp(f)$ and $R \cap V(f) = \varnothing$, by layering hypothesis.

For an And node $f = \bigwedge_{i=1}^{n} g_i$:

$$\begin{aligned} U(f) &= \prod_{i=1}^{n} U(g_i) \\ &\leq \prod_{i=1}^{n} 2^{|R \cap V(g_i)|} L_1(g_i) \\ &= \prod_{i=1}^{n} 2^{|R \cap V(g_i)|} \times \prod_{i=1}^{n} L_1(g_i) \\ &= 2^{\sum_{i=1}^{n} |R \cap V(g_i)|} L_1(f) \end{aligned}$$

and observing that $V(f) = \biguplus_{i=1}^{n} V(g_i)$:

$$= 2^{|R \cap V(f)|} L_1(f)$$

For an Ite node with variable in A, *i.e.* $f = \text{ite}(v, g, h)$, $v \in A$: $U(f) = \max(U(g), U(h)) \leq \max(L_1(g), L_1(h)) = L_1(f)$.

For a relaxed Ite node: $f = \text{ite}(v, g, h)$ with $v \in R$. $U(f) = U(g) + U(h)$. By induction hypothesis, $U(g) \leq 2^{|R \cap V(g)|} L_1(g) = 2^{|R \cap V(f) \setminus \{v\}|} L_1(g)$ and similarly for h. By summing:

$$\begin{aligned} U(f) &\leq 2^{|R \cap V(f) \setminus \{v\}|} \left(L_1(g) + L_1(h)\right) \\ &\leq 2^{|R \cap V(f) \setminus \{v\}|} \times 2 \times \left(\max\left(L_1(g), L_1(h)\right)\right) \\ &= 2^{|R \cap V(f)|} \left(\max\left(L_1(g), L_1(h)\right)\right) \\ &\leq 2^{|R \cap V(f)|} L_1(f) \end{aligned}$$

□

Given that fast and precise lower bounds improve over bad lower bound (Propositions 23 and 25), we obtain the same result for fast and precise bounds.

5.8.4 Summary

Relax (Definition 38) is therefore a parametric algorithm that behaves as Constrained (expensive compilation, exact result) without relaxed variables, as Unconstrained (relatively cheap compilation, loose approximation) when all chance variables are relaxed, but can also provide a trade-off between the two: the less relaxed variables there are, the more precise the answer, but the steeper the computational price.

5.9 Implementation & experiments

In this section we attempt to assess experimentally the tractability of f-**E-MAJSAT** on the kind of formulas required for quantitative robustness computation (Section 5.9.2), so as to illustrate the applications of quantitative robustness to vulnerability assessment (Section 5.9.3).

5.9.1 Popcon, a front-end for multiple f-E-MAJSAT algorithms

As the algorithms we presented are not always available as tools, and that those which are do not accept the same input format, we implemented our own front-end tool called Popcon. It accepts DIMACS or SMTLib2(QF_BV) input, converts this input to the appropriate format, including bitblasting (see Section 3.2.4) if necessary, and defers to an existing f-**E-MAJSAT** solver or a reimplementation when not available. We also added model counting and projected model counting abilities (based on D4 [82]) for comparison purposes. Popcon consists in about 8k lines of Rust. Bitblasting of SMTLib2 [10] input is performed by Boolector [97] down to AIGER format [14] and then by Tseytin transformation down to CNF. f-**E-MAJSAT** on the resulting CNF can be solved exactly with ssatABC (Section 5.7.3), dc-ssat (Section 5.7.1), Complan+ (Section 5.7.4), and Constrained (Proposition 18); or solved approximately with Maxcount (presented in Section 5.7.2), Oval (Section 5.7.4), and Relax (Definition 38).

Technical details All decision-DNNF-based algorithms are our own reimplementation, as neither Complan nor Complan+ are available as tools. decision-DNNF compilation is performed by the D4 [82] compiler. We also considered C2D [42] and Dsharp [94], but C2D cannot generate layered formulas and is thus only usable for model counting, and we hit bugs in Dsharp. We use a slightly patched version of D4 to disable some optimizations to the projected model counting of D4 that make its original layered decision-DNNF output not equivalent to the input formula. As Oval only provides an upper bound, we complement it by one of the lower bounds of Section 5.8.2, fast bound unless otherwise specified.

For Maxcount, ssatABC and dc-ssat, Popcon defers to the original tools, which return a floating point satisfiability probability. As our formulas are obtained through Tseytin transformation, they contain many variables determined by the value of other variables, and have thus a very low satisfiability probability, and the answer frequently underflows. We therefore patched ssatABC to compute an exact answer with GMP rational implementation. Other algorithms did not appear competitive enough in their inexact form to warrant similar porting effort. We use a patched version of dc-ssat that inputs a SSAT formula instead of a probabilistic planning domain kindly provided by N.-Z. Lee and used in the performance evaluation of ssatABC [85]. In the case of Maxcount, we set the number k of clones of the formula to 1, which lowers precision to obtain better performance.

Relaxation Popcon provides an implementation of Relax (Section 5.8) by asking D4 for a $(A \uplus R, X)$-layered decision-DNNF formula instead of a $(A, R \uplus X)$-layered one. Let us briefly discuss the choice of R. Popcon offers two ways to choose R under the constraint that $|R| \leq r$, where r is a user-controlled parameter:

DFS(r) In this mode, we rely on the decision heuristics of D4 during compilation. Compilation starts with $R = \varnothing$. We patch D4 to add variables it would have decided if not constrained to R until $|R| = r$. In other words, R contains the first r variables the compiler wants to decide. D4 operates in depth-first search order, hence the name;

BFS(r) In this mode we try to mimic the of decisions of model counting by running D4 for model counting, and collecting the r top-most decided variables in breadth-first-search order in the resulting decision tree.

Once Popcon obtains the required normal form, it can compute bad, fast or precise bounds (Section 5.8.2).

Correctness (sanity check) As part of the test suite of Popcon, we compare the results of the decision-DNNF-based algorithms we reimplemented to brute force enumeration of random 3-SAT formulas with 19 variables and 76 clauses (small enough to make brute force tractable). We also compared the result of **f-E-MAJSAT** on larger instances with different underlying decision-DNNF compilers (D4 and Dsharp). This testing has been intensive enough that it surfaced bugs in these compilers. Overall, this makes us confident that these algorithms are reasonably correct.

The port of ssatABC to GMP passes ssatABC's own test suite. It was profiled and does not spend significant time in GMP code, which indicates that performance assessments should not be skewed by the cost of rational computations.

5.9.2 Experimental evaluation

We conducted experiments to answer the following research questions:

RQ1 Can **f-E-MAJSAT** on the formulas generated during QRSE be solved exactly in practice, and how do the various algorithms we described compare?

RQ2 Do the approximate algorithms we described allow to solve more instances, and at what cost in terms of precision?

RQ3 Focusing on Relax, how do the various parameters influence performance and precision?

RQ4 We argued in Chapter 4 that quantitative approaches would be significantly more expensive than the qualitative approach of robust reachability. How do **f-E-MAJSAT** and universally quantified SMT solving compare?

RQ5 Can we venture explanations for the relative poor performance of some techniques as shown in **RQ1** and **RQ2**?

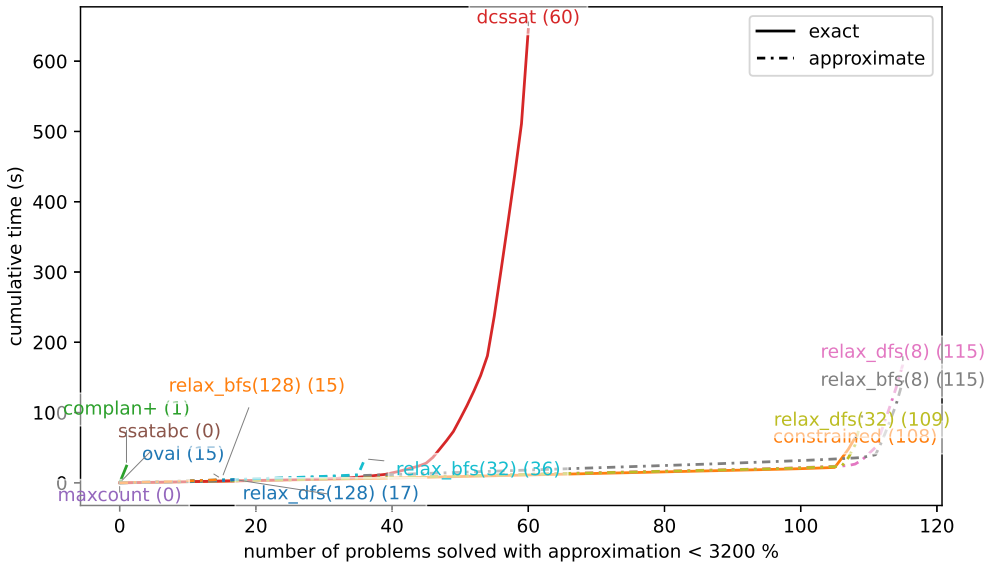
To answer these questions, we prepared a benchmark composed of 117 samples:

RSE 92 SMTLib2 formulas obtained by RSE on the case studies of Section 4.6.2.1;

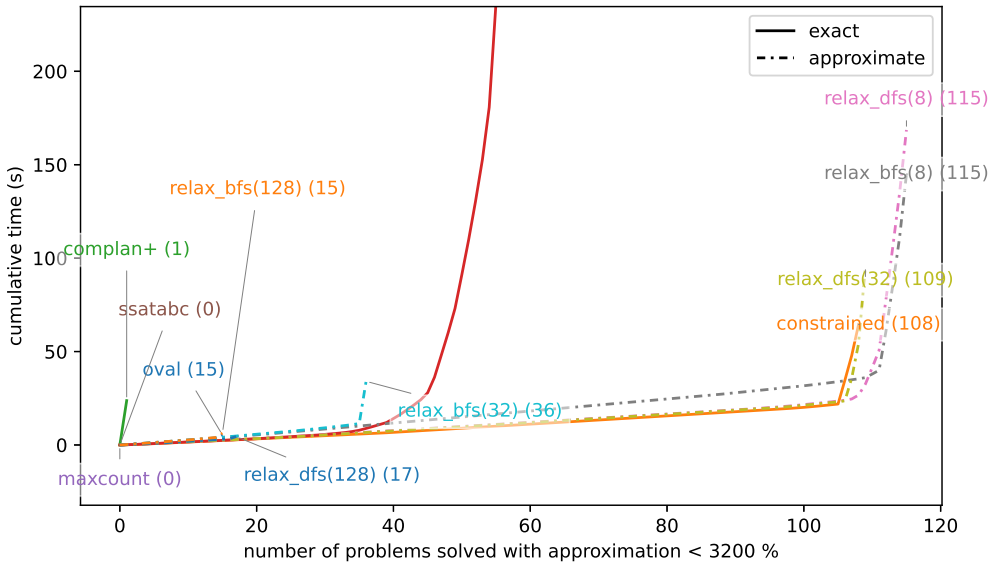
VerifyPIN The 25 distinct SMTLib2 **f-E-MAJSAT** problems generated during our case study about VerifyPIN (Section 5.9.3.2).

The size distribution of these formulas is described in Table 5.3. It is comparable to what is found in Lee, Wang, and Jiang [85] (331 variables and 3761 clauses in median). Problems are run on an Intel Xeon E-2176M CPU (2.70GHz) with a timeout of 20 minutes and memory-out of 2 GB.

RQ1 As can be seen in Figure 5.4, only two exact methods can solve a significant number of instances. dc-ssat solves about half the instances (60/117) and Constrained 108/117. This is surprising because Complan+, which only solves 1 instance, was designed to improve over Constrained, relying on the fact that compilation to decision-DNNF is significantly more expensive when constrained than when unconstrained. This assumption is true: Oval, which



(a) Full cactus plot



(b) Detail

Figure 5.4: Cactus plot of various f-**E-MAJSAT** solving algorithms on 117 instances coming from QRSE. Dashed lines correspond to methods returning an interval rather than an exact answer. Number of solved problem instances is given in parentheses. Only instances where imprecision is below $32\times$ (*i.e.* the reported interval $[l, h]$ satisfies $h \leq 32l$) are counted as solved.

Table 5.3: Size distribution of the formulas used for this benchmark

	Minimum	Median	Maximum
Clause count	508	998	58307
Variable count	247	554	19901
Problem variable count	80	226	570

Problem variables are variables which were not introduced by Tseytin transformation as part of bit-blasting. Variables introduced by Tseytin transformation are fully determined by the value of problem variables, and thus contribute less the complexity of compilation.

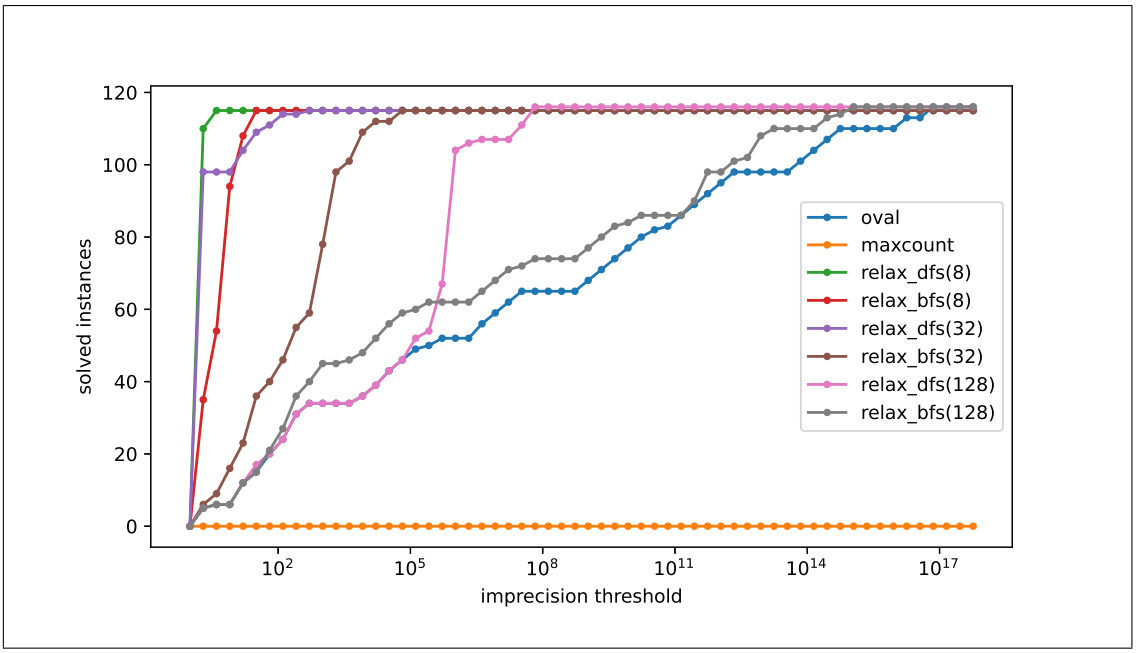
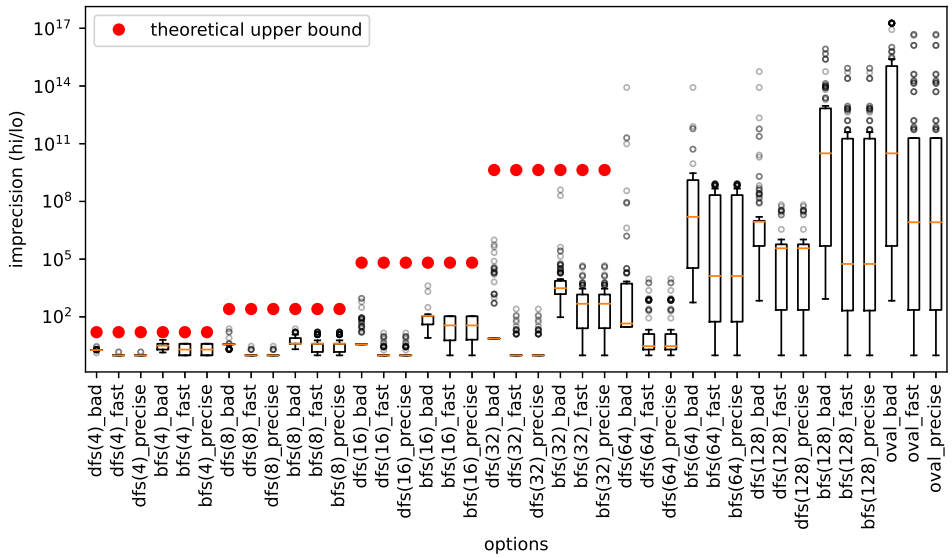


Figure 5.5: Evolution of the number of instances solved under a threshold of precision. Only approximate methods are represented.



Theoretical upper bound (Theorem 2) 2^r omitted for $r \geq 64$.

Figure 5.6: Imprecision of approximate f-**E-MAJSAT** solving algorithms, computed as upper bound divided by lower bound of the interval.

can operate on an unconstrained decision-DNNF solves 8 more instances than Constrained when one does not care about the precision of the result (see Figure 5.5). This implies that the relative poor performance of Complan+ comes not from decision-DNNF compilation but from the branch and bound step, which repeatedly performs a counting procedure on the decision-DNNF. Similarly, ssatABC solves no instance.

Constrained is the only exact algorithm which performs well on formulas generated by RSE (even better than Complan+, which was designed to improve on it), and it still leaves 7% of instances unsolved.

RQ2 To solve more than 108/117 instances one needs to resort to approximate techniques, which return an interval $[l, h]$ for the result. Oval can solve all instances but one, and Relax can solve from 114 to 116 instances depending on parameters. But these results are quite misleading as they do not take into account the quality of the answer. We call the imprecision of the technique the ratio h/l. The cactus plot of Figure 5.4 shows the number of solved instances under an arbitrary threshold of 32×, but Figure 5.5 summarizes results for other imprecision thresholds. One sees that Oval provides quite bad approximation, while Relax can solve 115 instances with imprecision under 4× with 8 relaxed variables. Finally, Maxcount always times out.

*Approximate f-**E-MAJSAT** algorithms allow solving more instances, and Relax can do so while preserving good precision: 115/117 instances solved instead of 108/117 exactly while keeping under a factor 4 of approximation.*

RQ3 As can be seen Figure 5.7, the number of instances solved by Relax within timeout indeed increases with the number r of relaxed variables. Up to 8 more instances can be solved with relaxation. The imprecision also increases with r, as shown in Figure 5.6, but it is orders of

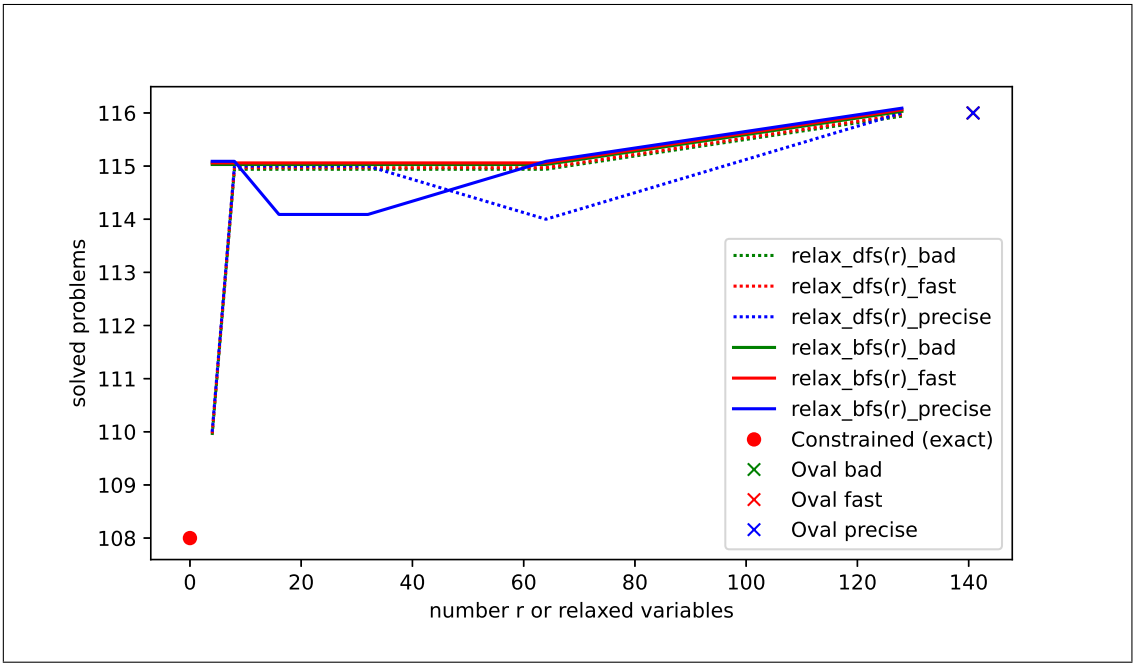


Figure 5.7: Efficiency for various relaxation parameters: number r of relaxation variables, and algorithm for the lower bound.

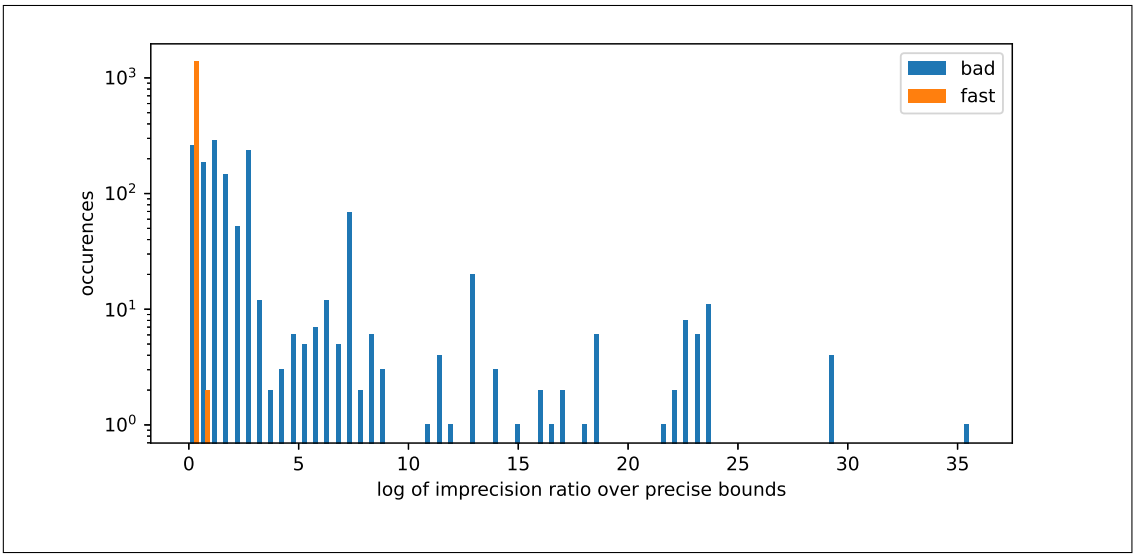


Figure 5.8: Ratio of imprecision for fast and bad bounds over imprecision for precise bounds.

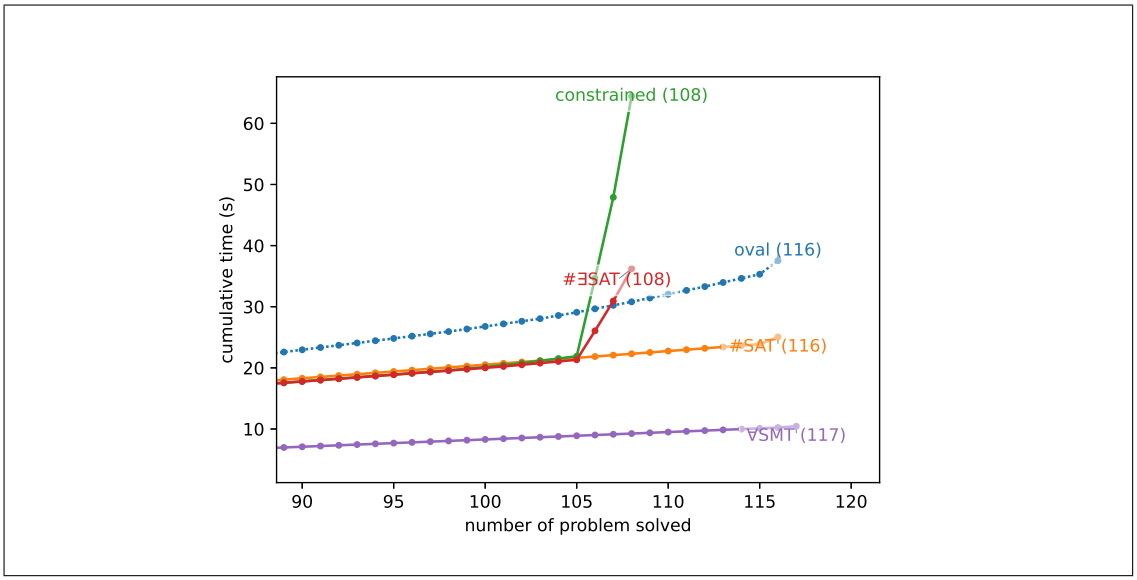


Figure 5.9: Comparison of the cost of solving f-**E-MAJSAT** to universally quantified SMT, model counting and projected model counting.

magnitude smaller than the theoretical bound 2^r (Theorem 2). DFS variable order usually yields more precise results, but for high r values (128) the tendency inverts in median. As expected (Proposition 27), when r becomes large, one obtains similar behavior as techniques based on fully unconstrained decision-DNNF, like Oval. As visible in Figures 5.6 and 5.8, bad lower bound is significantly lower than fast and precise bounds, which are very close to one another. Precise lower bound is more precise than fast lower bound in only two instances, but it does cost one more timeout in some cases because it has quadratic complexity instead of linear. One can legitimately conclude that fast lower bound has a better cost/effectiveness ratio.

*Relaxation allows to find a sweet spot in terms of precision and efficiency which solves more instances than exact f-**E-MAJSAT** with significantly better approximation than (worst-case) theoretical bounds.*

RQ4 For this research question, we consider our test suite as a set of SMT formulas comprising a partition of controlled variables a and uncontrolled variables x, a set of assertions interpreted as their conjunction $\mathrm{pc}(a, x)$, assumptions about controlled variables $h_a(a)$, and assumptions about uncontrolled variables $h_x(x)$. We previously focused on solving f-**E-MAJSAT** on $h_a(a) \wedge h_x(x) \wedge \mathrm{pc}(a, x)$ as per the definition of quantitative robustness (Equation (5.3)). We now compare to what we would have had to solve with robust reachability: we denote as ∀**SMT** the problem of determining the satisfiability of the corresponding formula generated by RSE: $\exists a.\, h_a(a) \wedge (\exists x.\, h_x(x) \wedge \forall x.\, h_x(x) \implies \mathrm{pc}(a, x))$. For informational comparison purposes, we also include model counting ♯**SAT**: $\sharp\,(h_a(a) \wedge h_x(x) \wedge \mathrm{pc}(a, x))$ and projected model counting ♯∃**SAT**: $\left|\left\{a \mid h_a(a) \wedge \exists x.\, h_x(x) \wedge \mathrm{pc}_{a,x}\right\}\right|$. Model counting and projected model counting are solved by D4 [82], and ∀SMT by Z3 [48]. f-**E-MAJSAT** solving is represented by Constrained (exact) and Oval (faster, but imprecise).

Results are shown in the cactus plot of Figure 5.9. One can see that solving ∀**SMT** is 7 times faster for 108 instances than exact f-**E-MAJSAT**, and does not suffer from timeouts. Constrained times out 9 times, in comparison. Even when completely overlooking the quality

of the result, the inexact algorithm OVAL is still about 4 times slower, and has one time-out. *Overall, quantitative treatment of path constraints generated during RSE is indeed significantly more expensive than the corresponding qualitative treatment.*

As an aside, let us observe that CONSTRAINED is close to D4's implementation of projected model counting. This is not a coincidence as they are both based on constrained decision-DNNF. Similarly, OVAL and D4's implementation of **♯SAT** are both based on unconstrained decision-DNNF, and have similar performance profiles.

RQ5 In this research question, we seek to give more details to understand why the majority of existing algorithms to solve f**-E-MAJSAT** we compared in the previous experiments performed quite bad.

First, let us mention that DC-SSAT and MAXCOUNT were designed for more general problems than f**-E-MAJSAT**: we use them on a very specific corner case of their input space. Specifically, DC-SSAT is designed for formulas with many quantifier alternations, and performs divide-and-conquer on quantifier blocks: on our instances with only two quantifiers, there is not much to divide and conquer. MAXCOUNT is designed for additional projection: $\exists a. \text{Ƃ}^{\frac{1}{2}} x. \exists z. f(a, x, z)$. On a sample of size 1, it seems that MAXCOUNT relies on projection for performance: consider the one formula f provided with the source of MAXCOUNT to check that the installation is correct. MAXCOUNT solves $\exists a. \text{Ƃ}^{\frac{1}{2}} x. \exists z. f(a, x, z)$ with $k = 2$ in about 18 seconds, but when mapping it to proper f**-E-MAJSAT** ($\exists a. \text{Ƃ}^{\frac{1}{2}} x, z. f(a, x, z)$) it is still running after 20 minutes, even with the more performance-friendly setting of $k = 1$.

We now turn to SSATABC and COMPLAN+. As they solve 0 to 1 instances in our previous experiment, we will use simpler formulas: $f_n(a, x) \triangleq x <_{u,n} a$ where $<_{u,n}$ represents the unsigned lower than operator on n-bits bitvectors. To investigate the scalability of these techniques, we want to determine the amount of work they perform depending on n from 2 to 24. SSATABC and COMPLAN+ have in common that they try a number of values of a until they can prove (by clause selection or branch-and-bound) that they found the optimal one. The number of values of a tested is a good proxy for the amount of work they perform as SSATABC performs two SAT solvings and a few model countings per value of a, and COMPLAN+ performs one computation of OVAL per value of a. We compare this to techniques that count once on decision-DNNF: CONSTRAINED, RELAX with BFS(20) and fast bounds, and model counting. All these compile to decision-DNNF and perform a linear algorithm on it, so the size (in nodes) of the decision-DNNF is a reasonably good proxy for the amount of work the algorithm needs. To compare fairly, we normalize this amount of work to 1 for size 8. As can be seen in Figure 5.10, SSATABC, COMPLAN+ and CONSTRAINED take an exponential amount of work. Each individual "work step" can cost a different amount of time for these three algorithms, which explains the results of **RQ1**, but in the end they do not scale very well. On the other hand, model counting (relying on unconstrained decision-DNNF) and RELAX (relying on relaxed decision-DNNF) scale almost linearly. Note that in these experiments, RELAX always has imprecision below 2.2× and returns in less than a second.

Finally, we argued that these tools were possibly tuned to different kinds of formulas. We can replay the same experiment that we did before for **RQ1** and **RQ2** on formulas coming from the test suite of SSATABC instead of QRSE. It is composed of 165[4] CNF formulas coming from various domains like probabilistic planning, quantitative information flow, *etc.* We will not go into as much detail as previously but results are summarized in the cactus plot of Figure 5.11. One can see that on the kind of formulas SSATABC was designed for it is much more efficient, solving 147/165 instances, whereas decision-DNNF-based techniques do not solve more than

[4]We excluded randomly generated formulas.

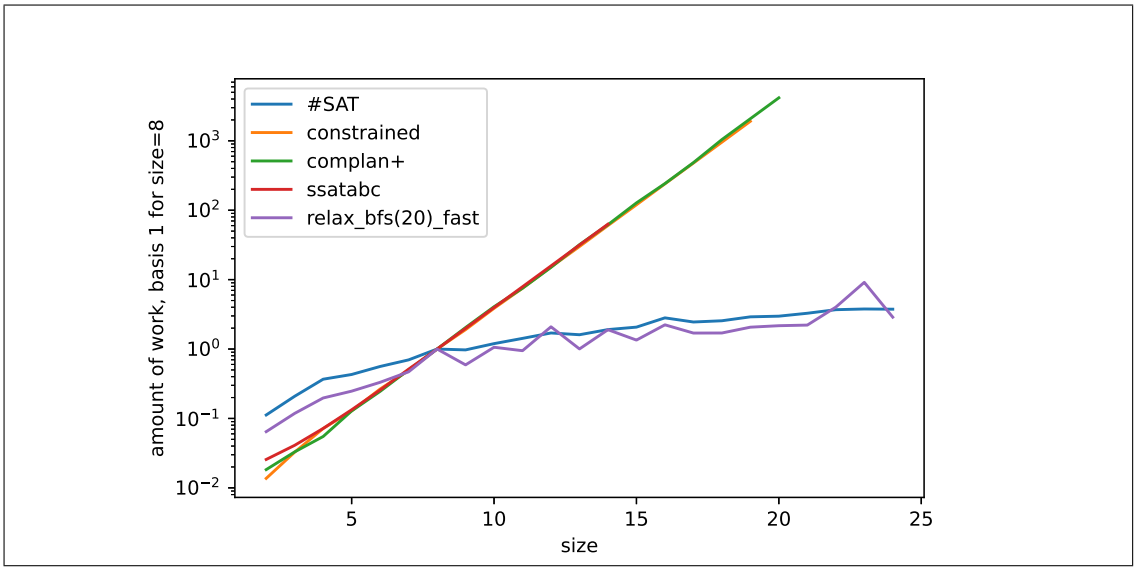


Figure 5.10: Comparison of the scalability of various algorithms for f-**E-MAJSAT** on $x < a$ with x, a bitvectors of size n. The amount of work needed by each algorithm is measured as number of iterations for ssatABC and Complan+, and number of nodes of the decision-DNNF for other algorithms, normalized to 1 for size 8.

100/165 instances. Relax (less than 77/165) is even worse than Constrained (100/165) in terms of number of instances solved, because we only count instances solved sufficiently precisely (imprecision below 32×) as successfully solved. However, we keep the surprising result that Complan+ (43/165) performs worse than Constrained (100/165) which it was supposed to improve upon. It is good to remind the reader here that the formulas generated by QRSE have a significantly different shape than, for example, those generated from probabilistic planning: they are bitblasted from SMTLib2 input, have very low f-**E-MAJSAT** and variables represent bits or logical gates, whereas in probabilistic planning bits represent the actual state of the system without indirection through low-level bit representation in the compiled program and then bitblasting.

Unconstrained and relaxed decision-DNNF distinguish themselves from ssatABC, Complan+ and Constrained by avoiding exponential blow-up on bitblasted formulas from the bit-vector theory of SMTLib2. This advantage seems to disappear on formulas coming from other domains, where other algorithms perform better.

Summary We showed that on a set of f-**E-MAJSAT** instances stemming from QRSE, Constrained is the only exact algorithm which performs well (even better than Complan+, which was designed to improve on it), and it still leaves 7% of instances unsolved. Approximate f-**E-MAJSAT** algorithms allow solving more instances, and Relax can do so while preserving good precision: 115/117 instances solved instead of 108/117 exactly while keeping under a factor 4 of approximation. Relaxation allows to find a sweet spot in terms of precision and efficiency which solves more instances than exact f-**E-MAJSAT** with significantly better approximation than (worst-case) theoretical bounds.

Unconstrained and relaxed decision-DNNF distinguish themselves from ssatABC, Complan+ and Constrained by avoiding exponential blow-up on bitblasted formulas from the

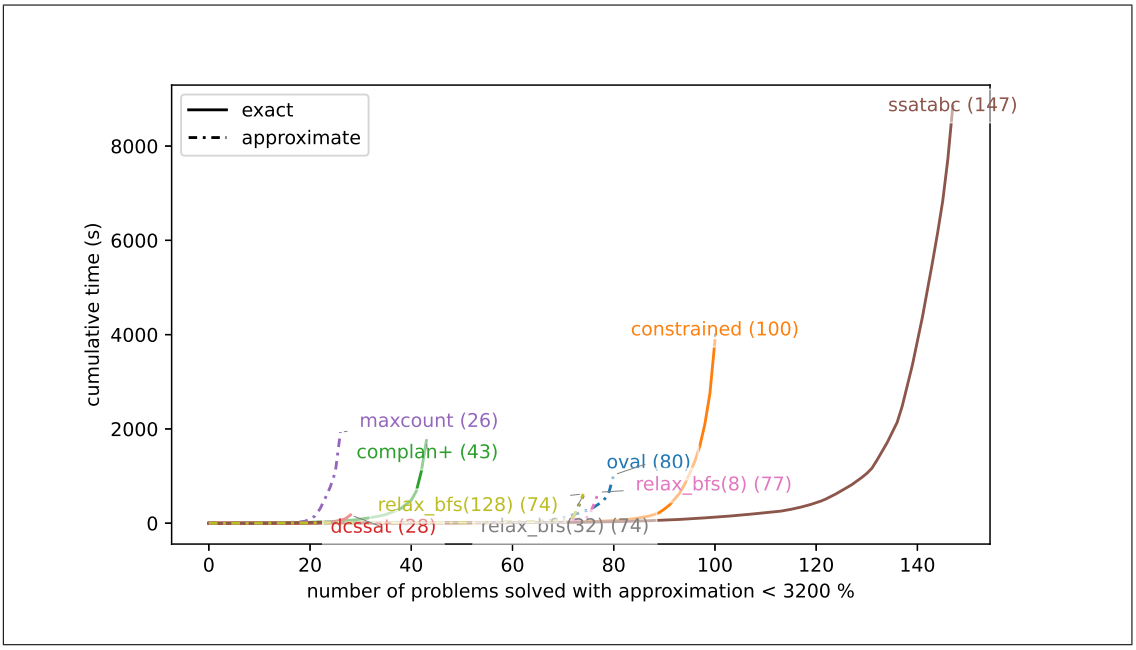


Figure 5.11: Cactus plot of various f-**E-MAJSAT** solving techniques on the test suite of SSATABC [85]. Again only instances solved with imprecision below 32× are counted as solved.

bit-vector theory of SMTLib2. This advantage seems to disappear on formulas coming from other domains, where other algorithms perform better.

Coming back to the original issue that made us investigate f-**E-MAJSAT**, we show that quantitative treatment of path constraints generated during QRSE is indeed significantly more expensive than the corresponding qualitative treatment in RSE.

5.9.3 Case studies

We now turn to illustrating the usage of QRSE in vulnerability assessment.

5.9.3.1 Quantitative robust symbolic execution implementation

We modified the binary-level symbolic execution tool BINSEC/RSE (Section 4.6.1) to perform QRSE. For each candidate path, first a satisfiability test is performed, with Z3 [48]. Unsatisfiable paths are discarded. A universal satisfiability test is also performed with Z3 again; if the path is robust then quantitative robustness is 1. If the path is neither robust nor unsatisfiable, then the path constraint is simplified [55]. If controlled variables are simplified out, the f-**E-MAJSAT** problem behind quantitative robustness reduces to standard model counting, which we solve with d4 [82]. Otherwise, the f-**E-MAJSAT** instance is solved with our tool Popcon, with the CONSTRAINED algorithm unless explicitly said otherwise. As BINSEC generates formulas in the theory QF_ABV, we blast arrays before handing them to Popcon.

Our tool does not support arbitrary probabilistic distributions for uncontrolled inputs, only uniform distributions, but it is possible to specify their domain as intervals, or even with free-form assumptions. For example, it allows specifying ASLR for the initial value of the stack register *esp* as $esp \in [\texttt{0xaaaa}, \texttt{0xbbbb}]$ (where the interval is platform-dependent) and **assume** $esp\%16 = 0$ (alignment).

5.9.3.2 Triaging fault injection attacks in VerifyPIN

As an illustration of the potential usage of quantitative robustness in security analysis, let us consider the case of physical fault injection [60], where an attacker tries to subvert a high-security component (e.g., a smart card) through physically-induced runtime errors. We consider the program VerifyPIN (specifically, VerifyPIN_2) from FISSC [52], a standard benchmark in the fault injection community. VerifyPIN is a procedure mimicking the code checking the PIN entered for example on an ATM, including security-related countermeasures. It has two explicit inputs: the 4-byte entered PIN code (userPIN) and the PIN code stored on the card (cardPIN), and returns whether they are equal or not. For the sake of illustration, we adopt a threat model where an attacker controls the userPIN only[5], and can prevent the processor from executing one single instruction, effectively replacing it by `nop` (skip). The security question is *"Can such an attacker enter a PIN distinct from the cardPIN and still be granted access?"*. We applied the 126 possible 1-byte and 2-byte wide `nop` faults on VerifyPIN, obtaining 126 *mutants* (i.e., variants of the initial program emulating the considered hardware faults), and use variants of symbolic execution to find potential attacks.

Unfortunately all attacks are not equally threatening: for some of them, there is one single unduly accepted PIN, whereas for other ones, one of the four bytes of the cardPIN is not taken into account. We want to detect the more concerning attacks while sorting out the less replicable ones. We compare the 4 following approaches experimentally:

SE the SE implementation of BINSEC [49].

RSE the RSE implementation of BINSEC (Section 4.6.1).

exact QRSE A modified version of BINSEC/RSE using exact quantitative robustness. It is based on Constrained as it is the most effective exact algorithm in Section 5.9.2.

relaxed QRSE same as above but where we accept an approximate answer. Given the results of our benchmark of Section 5.9.2, we use Relax, and to get the best possible answer, we first try with $BFS(8)$ for half the timeout (because it provides tight bounds), and if this fails, with $BFS(128)$ with half the timeout (because it times out least often).

To compare qualitative and quantitative approaches fairly, we use a threshold: we attempt to identify traces which are above 20% (highly concerning) or below 10^{-6} (noise). For relaxed QRSE, we report traces *provably* in one of the category above – based on the reported lower and upper bounds. BINSEC and the SMT solver have no timeout, but Popcon is limited to 3 min.

Normal fault-based vulnerability analysis is normally performed in two steps: first an automated analysis like SE finds mutants admitting attack traces, *i.e.* one input leading to unexpected behavior, and then these traces are handed to experts for manual analysis. We are interested in reducing the amount of manual work needed by limiting the number of interesting traces. The thresholds mentioned above, while somewhat arbitrary, are chosen to illustrate two approaches: a conservative analysis where only traces with a provably low quantitative robustness are dismissed, and a more opportunistic one where one only analyzes provably concerning traces with high quantitative robustness.

As shown in Table 5.4, SE finds 39 attack traces, RSE finds none, and quantitative approaches find an intermediate number of them depending on the threshold. Exact QRSE has 13 timeouts, but still proves that out of the 39 attacks found by SE, at least 23 are not interesting ($< 10^{-6}$). Relaxed QRSE improves significantly in this regard, as there is no timeout when using the hybrid BFS(8) then BFS(128) approach[6]. It classifies 27 traces as not interesting, and finds

[5]Other inputs are uncontrolled: the userPIN, but also implicit input, e.g. uninitialized values accessed due to faults.

[6]Only BFS(8) has one timeout, and BFS(128) has no timeout but fails to classify many traces. Doing the

Table 5.4: Comparison of various methods to look for exploitable faults

Method	Quantitative robustness	Reported attack traces	Time (s)	Paths abandoned because of Z3 UNKNOWN	Paths abandoned because of Popcon timeout
SE	>0%	39	66	0	–
RSE	= 100%	0	67	25	–
exact QRSE	$> 20\%$	0	2435	0	13
	$< 10^{-6}$	23			
	$\in [10^{-6}, 20\%]$	3			
	unclassified	0			
relaxed QRSE BFS(8) then BFS(128)	$> 20\%$	2	250	0	**0**
	$< 10^{-6}$	27			
	$\in [10^{-6}, 20\%]$	10			
	unclassified	0			
relaxed QRSE BFS(8)	$> 20\%$	2	333	0	1
	$< 10^{-6}$	27			
	$\in [10^{-6}, 20\%]$	9			
	unclassified	0			
relaxed QRSE BFS(128)	$> 20\%$	2	67	0	**0**
	$< 10^{-6}$	21			
	$\in [10^{-6}, 20\%]$	6			
	unclassified	10			

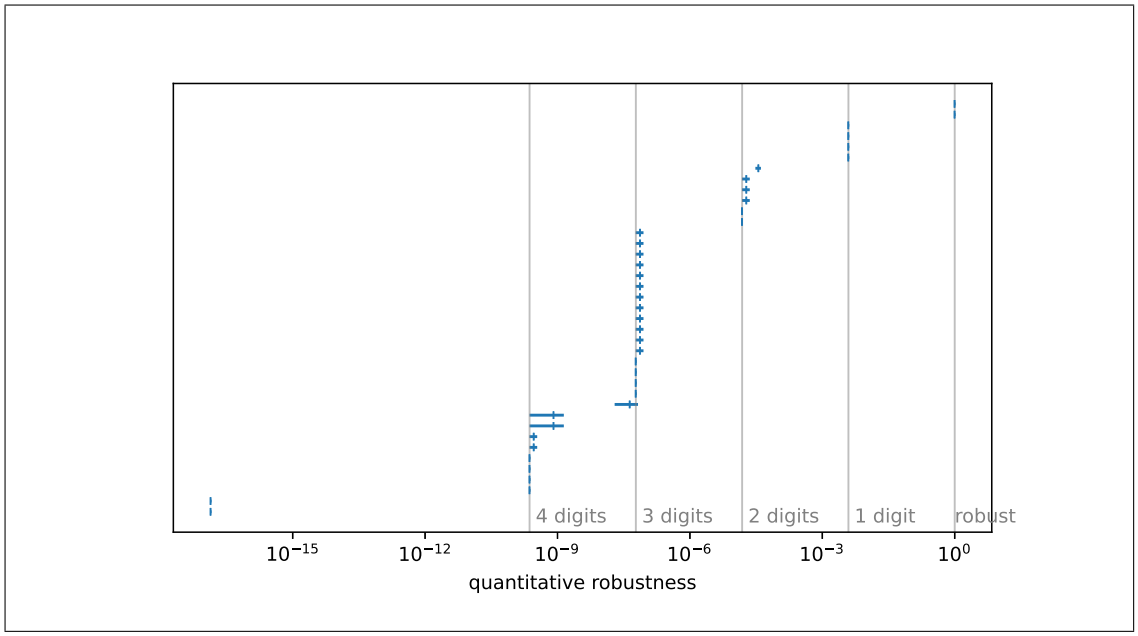


Figure 5.12: Distribution of quantitative robustness of attacks by `nop` faults on VerifyPIN obtained by relaxed QRSE. Error bars represent the interval computed by relaxation.

two concerning traces with quantitative robustness in $[0.992202, 0.992204]$. Figure 5.12 gives the distribution of quantitative robustness. Most traces are exploitable for about 1 case out of 2^{24} which corresponds to faults where only 3 bytes (24 bits) of the PIN out of 4 are correctly compared. The exact mechanism varies depending on faults, but usually looks like "the program does not load the next digit from memory and thus compares the i-th digit of the userPIN to the $i-1$-th digit of the cardPIN". Therefore, the attacker does not have to guess the i-th digit of the cardPIN. Some instances have very low quantitative robustness: 2 attack traces only happen for about 1 initial condition out of 2^{56}. Manual analysis shows that the attacker must guess 3 bytes of the cardPIN, the low byte of a register and hope for the top 3 bytes to be zero. Overall this amounts to 7 bytes, or 56 bits, of luck. Interestingly, the 6 top faults detected are outside the protected code of VerifyPIN. They affect the logic checking the result of VerifyPIN to accept or reject attacker input: indeed if one nops out the test "if VerifyPIN rejects userPIN then goto fail", the attacker wins for a large proportion of initial conditions. Therefore, regarding the actual protected part of VerifyPIN, relaxed QRSE proves that none of the 80 mutants on the protected part of VerifyPIN suffer from any attack with quantitative robustness above 10^{-4}.

In the end, this analysis allows to reduce the number of cases to analyze manually from 29 with standard SE to 12 in the conservative scenario described above, and to 2 in the optimistic one. It is also interesting to note that for the analysis we just made, the approximate f**-E-MAJSAT** algorithm we used ended up quite precise, as illustrated by Figure 5.12. Overall, QRSE proves useful here to help focus the attention of the security expert on possibly critical attack traces, and remove noisy ones.

inverse composition (BFS(128) and if it is not precise enough for Table 5.4 then BFS(8)) obtains the same classification as BFS(8) then BFS(128) faster (97 seconds) and with no timeout/unknown but yields larger error bars in Figure 5.12.

Table 5.5: Detecting CVE-2019-20839

Method	SE	exact QRSE >20%	RSE no path merging	RSE with path merging
Detected	✓	✓	✗	✓
Guarantee on quantitative robustness	> 0 %	25 %	–	100 %
Total time (s)	997	1029	–	1254
Simplification time (s)	260	282	–	514
Z3 time (s)	718	723	–	711
Popcon time (s)	–	1	–	–

5.9.3.3 CVE-2019-20839 in libvncserver

We consider a stack buffer overflow in libvncserver. The security question is: *Can an attacker controlling the address of the server use this stack buffer overflow to divert control flow to* `0xdeadbeef`*?* Standard SE tells us it is possible for example when the top of the stack is at `0xfff02000` and various other initial conditions are met. But all of those, except the arguments, are beyond the control of the attacker, making this information of little use for vulnerability assessment. RSE can prove the stronger robust reachability: by choosing the right server address, the attacker can trigger the buffer overflow for all initial conditions. However, this requires systematic path merging, which is documented to be useful when used carefully but detrimental to performance when used indiscriminately [69, 80]. The necessity of path merging in this instance comes from the fact that the optimal attacker reaches the target with one of 3 paths in an uncontrolled way. No single path satisfies robust reachability, but still one of them at least must have a large quantitative robustness: at least $\frac{1}{3}$ (see Section 5.5.2 where we treated this formally). In other words, instead of using path merging, we can attempt to detect single paths with high quantitative robustness. This indicates that an optimal attacker can trigger the bug for a high proportion of uncontrolled inputs, which is weaker than robust reachability but still a good hint of security relevance.

Table 5.5 is a summary of our results. We see that here our quantitative robustness-based solution makes path merging unneeded and is faster than (qualitative) RSE+. Solving ***f*-E-MAJSAT** only takes 1 second, and allows manipulating simpler path constraints which take significantly less time to simplify. This is a counter-intuitive case where our original quest for more precision at the expense of efficiency ends up with a faster technique with comparable results.

This case study illustrates that to some extent, our quantitative approach can reduce the need for path merging in robust symbolic execution, and more generally allows us to miss some behaviors (for example because of the lack of stubs for I/O, system calls, *etc.*) while still obtaining valuable information on the replicability of the bug. We even get the good surprise of improved performance.

5.9.3.4 Summary

In a case study about VerifyPIN, we illustrated the usage of QRSE as a form of bug-finding capable of bug triage according to quantitative robustness. QRSE gives much more insight on the security impact of bugs found in this case because in the domain of fault injection it is rare to find flaws that give a 100% chance of winning to the attacker. One wants to detect faults

which give even a comparatively smaller edge to the attacker, and those are invisible to robust reachability.

We presented a second use case in libvncserver: quantitative robustness is better behaved than robust reachability in the sense that it cannot vanish entirely at branches; as a result one does not need path merging so much. Surprisingly, omitting path merging even improves performance compared to robust symbolic execution even though we resort to comparatively expensive model counters.

5.10 Related work & discussion

We attempt at designing a quantitative counterpart to robust reachability, viewed as too strict. Such a quantitative relaxation has already been seen in other domains and is part of a general effort to make formal verification less "all-or-nothing": from non-interference [65] to quantitative information flow [73], from traditional model checking to probabilistic model checking [5, 70] or from symbolic execution to probabilistic symbolic execution [59].

These different applications give rise to different counting problems. Quantitative robustness relies on f-**E-MAJSAT** to distinguish between attacker-controlled inputs and uncontrolled inputs, while probabilistic verification builds on standard model counting [67] and quantitative information flow on projected model counting [6].

Model counting as a whole has seen a lot of recent research in different directions: beyond compilation-based approaches [82, 94, 42], we can cite DPLL-based model counters [117], probabilistic caching [112], approximate approaches based on XOR-constraints [66]. As formula compilation is a clear bottleneck of our technique, it is worth exploring how such alternatives could be adapted for f-**E-MAJSAT**. Notably XOR-constraints, which have been very successful in model counting, have an adaptation for Bayesian networks [122], which would deserve to be adapted for f-**E-MAJSAT**.

Regarding approximated solutions to f-**E-MAJSAT**, many combinations and extensions are possible. The branch-and-bound algorithms behind Complan and Complan+ can be interrupted at any time to obtain a refined, but not perfect interval. Our algorithm Relax could be refined by using bounds inspired from Oval instead of Unconstrained, at the price of significant added complexity. Finally, the choice of the set of relaxed variables has only been partially explored, and is certainly a direction for future work.

Interestingly, some works target model counting problems beyond propositional formulas (e.g., for bit-vectors [77] or integer polyhedra [46]). That could be a source of inspiration for further developments.

Finally, while we discuss here security applications, robustness has other possible area of interest, such as flakiness analysis, as said in the previous chapter. Quantitative robustness has certainly a role to play there as well.

5.11 Conclusion

We introduce quantitative robustness, a quantitative counterpart to robust reachability, which expresses how easily an attacker can trigger a bug in a program, taking into account that he can only influence part of the program input. The asymmetry between attacker-controlled variables and variables out of his reach makes it an instance of a problem called f-**E-MAJSAT** [87], a distinct problem from the standard model counting arising when we are only interested in verification, without an attacker [59].

We assessed various existing solving techniques for the f-**E-MAJSAT** problems generated when computing quantitative robustness. Like for many hard problems, efficient solvers are usually directed towards some specific kinds of problem instances. As we used f-**E-MAJSAT** solving algorithms on a new kind of formulas arising from a new domain of application, we saw unexpected behavior: Complan+ [101] performs significantly worse than the basic algorithm it was designed to improve upon, Unconstrained. Overall, the best algorithms, based on knowledge compilation, can solve many, but not all, f-**E-MAJSAT** instances coming from QRSE.

To solve harder instances, we introduce a new approximate algorithm inspired from two existing decision-DNNF-compilation based approaches which enables to strike a compromise between precision and performance in a flexible and parametric way. We show experimentally that it allows going beyond existing algorithms in this regard, and that it obtains a much better precision than the theoretical bounds.

We finally illustrate on two case studies how the quantitative extension of robust reachability can be helpful for vulnerability assessment, quantifying how often the attack will succeed for optimal attacker input and thus giving valuable insight on its exploitability.

While our technique is already powerful enough to enable small realistic case studies, solving f-**E-MAJSAT** is currently a bottleneck toward scalability to larger applications. This is a clear direction for future work.

Chapter 6

Conclusion and future work

6.1 Conclusion

Bug-finding techniques inspired by works in the domain of verification, where all bugs are equally important and must be fixed, deserve to be revisited when we repurpose them for security-related goals. In this work we revisit reachability to account for the threat model, *i.e.* the capabilities of the attackers.

Specifically, we attempt at assessing how replicable bugs are. We observe that proving that a bug is reachable only tells us that there is an input which triggers it, but part of this input might be random or to the least out of the control of the attacker. In this case, this information is not enough to determine if an attacker could trigger the bug without relying on luck, somehow. As triggering the bug is the first step towards its exploitation, finding a reachable bug with classical bug-finding techniques leaves us clueless when it comes to its vulnerability assessment.

Therefore, we propose two refinements of reachability to assess how replicable the discovered bugs are. This allows to prioritize the bug-fixing effort on more easily replicable bugs first, which becomes all the more crucial as modern bug-finding are now so effective that in some case they can "drown" [78] the developers with innumerable reports. These refinements have in common that they take the threat model into account: some inputs to the program are labeled as "controlled" by the attacker and some others as "uncontrolled".

The first refinement is robust reachability, a property stronger than reachability, that expresses that there is a value of attacker-controlled input such that the bug is triggered for all values of uncontrolled inputs. A bug which satisfies this criterion can be triggered by the attacker without relying on luck, which is a strong hint of security impact. We propose practical proof methods that rely on the ability of state-of-the-art SMT solvers to handle universally quantified formulas. This helps avoid reports from *e.g.* symbolic execution about bugs which an attacker could hardly replicate. In this sense, this can be a great tool to help defenders focus the bug-fixing efforts on more valuable bugs first. Further, for bugs which an attacker can indeed reproduce but for which standard symbolic execution reports a non-robust trace, this allows obtaining a better bug report which should not mislead humans into underestimating the severity of the bug.

The drawback of the reliance on universal quantifiers of robust reachability is that it is "all-or-nothing". Bugs which an attacker could only trigger for 99% of uncontrolled inputs are ignored. Therefore, we refine our approach into a concept we call "quantitative robustness" and which corresponds to this value of 99%. This increase in precision comes at the price

of more computationally intensive proof methods, and we spend some time discussing solving algorithms and present our own. This somehow "unlocks" applications like vulnerabilities related to hardware faults where a fault is expected to give an edge to the attacker but certainly not be powerful enough to work 100% of the time. This kind of vulnerabilities is quite advanced and normally analyzed manually by experts, so once again, this technique can be viewed as a filter to reduce the amount of vulnerability reports.

6.2 Perspectives

We now present some directions to extend or improve the work we have presented.

First, robust reachability and quantitative robustness as we defined were designed with some intentional limitations, which we could attempt to lift.

Interactive systems The threat model we consider does not really fit interactive systems, because the attacker can only submit controlled input in one go without knowing anything of the system. This is to have only one quantifier alternation: $\exists \mathsf{controlled}.\, \forall \mathsf{uncontrolled}.\, \mathsf{bug}$. Each new round of interactivity adds a quantifier alternation.

The most simple extension of this is thus a form of parametric robust reachability where there is an additional parameter of the system, chosen by the environment but known to the attacker: $\forall \mathsf{parameter}.\, \exists \mathsf{controlled}.\, \forall \mathsf{uncontrolled}.\, \mathsf{bug}$. In the example of CVE-2019-20839 in libvncserver (Section 4.6.2.1), one would want to prove that for all possible return addresses not containing a null byte, the attacker can provide a controlled input which diverts control flow to this address. Superficial experiments suggest that while a naive encoding to the SMT solver would perform quite bad (Section 4.6.4.3), we could solve the case of libvncserver with a slightly smarter encoding based on the observation that the return address is directly encoded into the controlled input without transformation, making the outermost quantifier reasonably easy to eliminate. While this is a promising result, this technique not only limited to cases where the attacker only copies the parameter as-is, but also to cases where memory is fully simplified out by RoW [55]. Further work is needed to see whether these limitations can be lifted, or turn out to not be so much of a problem in practice.

Probabilities The tools we develop to compute quantitative robustness are designed with proportion of inputs in mind, not probabilities. If we insist in interpreting their output as probabilities, it means they are limited to the uniform distribution for uncontrolled inputs. What if we allowed more distributions?

The d-DNNF-based algorithms we took in the literature do support independent Bernoulli distribution for individual bits of input. The algorithm we contribute (Relax) probably extends easily to this case. On the other hand, supporting dependent bits looks like a much harder problem.

But one must not neglect the other half of the problem: we need to determine the probability distribution of the many architecture-dependent implicit inputs. It is known in cases like ASLR where randomness is intentionally introduced, but what about the initial content of registers, addresses returned by `malloc`, address of various low-level constructs like thread stacks, and so on? There is certainly some work in this regard.

Vulnerability assessment beyond replicability We assess the replicability of bugs, because the first step to exploit a bug is to trigger it. However, it is only one of the many dimensions

of vulnerability assessment, and some other directions deserve to be explored. To mention one which is related to what we presented, it would be interesting to quantify the amount of control that an attacker gains after the attack: in the case of a stack buffer overflow like CVE-2019-20839 in libvncserver (Section 4.6.2.1), one could want to know whether the overwritten return address only has 2 possible values or if it can take billions of distinct values. In the most naive formulation it is a form of projected model counting, and Binsec can prove that in the case of libvncserver there are more than 4 billions distinct possible return addresses. However, this does not account for the threat model (what inputs are controlled by the attacker and to what extent) and doing so would probably make the problem significantly more complex, just like with robustness we go from mere model counting to f**-E-MAJSAT**.

There are also some limitations to the techniques we presented that were not intentional but also deserve some exploration.

Brute force attacks A bug which is not replicable is not necessarily harmless. DirtyCow (CVE-2016-5195) is a vulnerability in the Linux kernel that allows privilege escalation via a race condition. The race condition is inherently flaky and thus not robust, but one can try it thousands of times. In practice, it takes seconds to exploit the bug on vulnerable systems.

This flaw in the concept of robust reachability can be addressed in two ways. On a formal level, the system that the attacker exploits is somehow not the vulnerable program, but the program repeated several times: the star of the program. Some of the formal discussions about robust reachability probably still hold if we consider the star of the program as the program to be analyzed. However, this is certainly impractical for proof methods.

The second possible direction is incorporating timing information to traces: if a bug has low quantitative robustness and that all traces reaching it take long execution time, then is becomes reasonable again to de-prioritize fixing it.

Model counting f**-E-MAJSAT** solving is the main bottleneck of QRSE, to the point that even for our motivating example (Chapter 2) we cannot get an exact value for all bugs. Many directions have been left unexplored.

Some of the most effective techniques in model counting are based on XOR-constraints, and it would be interesting to test a XOR-based f**-E-MAJSAT** solving algorithm (possibly adapted from Xue et al. [122]).

SMT solvers focusing on the theory of bit-vectors are much more efficient than first bitblasting then using a SAT solver. We could expect the same speedup by reasoning at this level for f**-E-MAJSAT**. Previous work has shown experimentally that a similar observation does not hold yet for model counting [18].

In our comparison of some existing solving algorithms we included exact and approximate algorithms. However, to some extent branch-and-bound-like algorithms (notably Complan+ and SsatABC) belong to both categories: they are exact but if you interrupt them before completion they yield an interval. We have not benchmarked the latter option.

Regarding our algorithm Relax, we show that two heuristics (BFS and DFS) used to select relaxed variables behave differently. It is possible that better heuristics, notably integrating within the decision-DNNF-compiler, would be even more effective.

Path merging in robust symbolic execution Similarly, we show that systematic path merging in robust symbolic execution can fail to prove some instances while merging only some paths according to a heuristic can succeed. There is some literature about wise heuristics for path

merging applied to standard symbolic execution; reevaluating it for robust symbolic execution may yield interesting results.

Robustness beyond security Finally, while we focused on the applications of robustness to security, we realized during this work that it also applied in other domain. Notably, it fits quite well the concept of flaky tests, a software engineering topic. A test is flaky when its outcome is non-deterministic because of an unexpected reliance on non-deterministic implicit inputs—alike to uncontrolled inputs. Flaky tests make continuous integration untrustworthy, as a test failure might just be a false negative, and one thus has to relaunch it to exclude this possibility. This increases time to feedback to the developer.

For example the path merging heuristic powering $\text{RSE}_{\forall}+$ relies on detecting branches with incoming robust path but outgoing non-robust paths. It seems that these are precisely branches where flakiness is introduced, suggesting it could be used to locate the sources of flakiness. This correspondence has been left unexplored. As illustrated in Section 4.6.2.2 one can also repurpose our tool to find tests parameters that fix the flakiness. More work in this direction might reshape this concept into a useful tool to improve the quality of test suites.

Annexe A

Résumé substantiel en français

Les méthodes modernes de recherche de bugs sont devenues si efficaces qu'elles détectent plus de bugs que les mainteneurs des logiciels affectés n'ont de temps à consacrer à les corriger. Notamment, des techniques telles que le fuzzing, dont l'efficacité repose sur le fait de ne pas analyser le programme sémantiquement, sont capables de détecter des centaines de défauts en peu de temps sans fournir ni analyse ni idée de correction. Les développeurs même les plus chevronnés peuvent alors se sentir « noyés » [78] par des rapports de bugs si nombreux qu'ils ne savent plus où donner de la tête.

Une posture communément adoptée pour faire face à cette problématique consiste à accorder aux bugs ayant un impact de sécurité, c'est-à-dire ceux dont un attaquant pourrait tirer parti, une sorte de priorité : les éditeurs de logiciels demandent à ce qu'ils soient rapportés différemment des autres, et s'engagent souvent à une réponse dans un délai déterminé, parfois explicitement plus court que pour les bugs de droit commun. Pour résumer de manière simpliste, les bugs n'ayant pas d'impact de sécurité ne méritent d'être corrigés que s'il reste du temps à leur consacrer.

Se pose alors la question de déterminer, parmi tous les bugs détectés par une méthode automatique, lesquels sont des *vulnérabilités*, c'est-à-dire ont un impact de sécurité. Cette question est vaste et difficile à formaliser ; au sens strict il y a une définition de vulnérabilité par modèle de menace. Nous nous intéressons dans cette thèse à une dimension qui contribue à l'impact de sécurité d'un bug : un attaquant pourrait-il déclencher ce bug ? Nous appelons cette propriété *réplicabilité*. Un bug que l'attaquant ne peut déclencher que lorsqu'il a de la chance est vraisemblablement moins grave qu'un bug qu'il peut déclencher avec certitude. Pour ne donner qu'un exemple, certaines classes de bugs comme les dépassements de tampon sur la pile (qui peuvent souvent donner à l'attaquant le contrôle total du programme) sont aujourd'hui protégées par randomisation : l'attaquant peut toujours les déclencher, mais uniquement s'il devine une valeur aléatoire de 32 ou 64 bits appelée canari [36]. Les méthodes de recherches de bugs telles que l'exécution symbolique sont toujours susceptibles de détecter de tels bugs, parce qu'effectivement sur le papier ils peuvent toujours arriver, mais dans la pratique nous en sommes protégés, et le rapporter est assimilable à un faux positif de fait.

L'enjeu de cette thèse est de concevoir des méthodes de recherche de bugs qui évitent ces faux positifs de fait, c'est-à-dire ne rapportent que des bugs qui sont suffisamment réplicables. Nous proposons deux approches allant dans cette direction : l'une qualitative et l'autre quantitative. Les deux reposent sur le même principe : on considère un modèle de menace où l'on distingue deux sortes d'entrées du programme : des entrées dites contrôlées (par l'attaquant), et des entrées dites non contrôlées (déterminées par l'environnement ou d'autres utilisateurs, ou même aléatoires).

L'approche qualitative consiste à définir une propriété appelée *atteignabilité robuste* qui exprime qu'il existe une valeur pour les entrées contrôlées telle que pour toutes les valeurs possibles des entrées non contrôlées le bug est déclenché. Autrement dit, lorsqu'un bug est atteignable de manière robuste, l'attaquant peut gagner sans se reposer sur la chance. Nous montrons qu'il est possible d'adapter l'exécution symbolique (et le bounded model checking) pour ne retourner que des bugs atteignables de manière robuste. Si l'on note les entrées contrôlées a et les entrées non contrôlées x, l'exécution symbolique traditionnelle détecte des bugs avec des requêtes de satisfiabilité modulo théories (SMT) de la forme $\exists a, x.\, \mathsf{bug}(a, x)$. Pour l'atteignabilité robuste on utilise des requêtes de la forme $\exists a.\, \forall x.\, \mathsf{bug}(a, x)$. Avec un quantificateur supplémentaire, elles sont plus difficiles à résoudre pour les solveurs SMT, mais nous montrons dans des études de cas portant notamment sur des rejeux de vulnérabilités réelles que cela ne nous empêche pas de remplir notre objectif initial : se débarrasser des faux positifs de fait.

Si le principe de base de l'exécution symbolique adaptée à l'atteignabilité robuste est simple (ajouter un quantificateur universel), d'autres adaptations s'avèrent nécessaires. Notamment, si l'exécution symbolique standard est presque complète pour l'atteignabilité standard au sens où elle est complète si l'on borne la longueur des chemins, cette propriété est perdue lorsqu'il s'agit de prouver l'atteignabilité robuste. Pour la retrouver, on peut recourir à la fusion de chemins, une technique connue en exécution symbolique standard comme une optimisation purement optionnelle, et qui devient ici obligatoire. Il est aussi nécessaire d'adapter d'autres choses telles que l'encodage des hypothèses, *etc.*

La limitation principale de l'atteignabilité robuste est son caractère tout-ou-rien : un bug tel que l'attaquant optimal ne peut le déclencher que dans 99% des cas ne sera pas considéré atteignable de manière robuste et par conséquent ignoré. Si notre propos est que c'est plutôt légitime lorsqu'il s'agit d'un cas sur 2^{32}, nous voudrions malgré tout détecter les bugs « à 99% ». Cela nous conduit à proposer une seconde approche quantitative.

On définit la robustesse quantitative comme la proportion d'entrées non contrôlées qui permettent à l'attaquant de déclencher le bug pour l'entrée contrôlée optimale. Cette valeur vaut 0 lorsque le bug n'est pas atteignable et 1 lorsque le bug est atteignable de manière robuste. La calculer nous donne accès à tous les cas intermédiaires « plus ou moins robustes », et l'utilisateur final peut placer le seuil de détection là où il le désire. L'extension de l'exécution symbolique à l'atteignabilité robuste s'adapte bien à la robustesse quantitative pourvu que l'on sait déterminer la proportion maximale de x telle qu'une formule $f(a, x)$ est satisfaite pour la valeur de a optimale. Ce problème, apparenté au comptage de modèle, s'appelle *functional* ***E-MAJSAT***. Il est loin d'avoir atteint le niveau de maturité des solveurs SMT que nous utilisions pour l'atteignabilité robuste. Les méthodes de résolution existantes ont été développées pour d'autres types de problèmes comme l'inférence de réseaux bayésiens et nous montrons expérimentalement que des méthodes avancées censées améliorer d'autres plus naïves se révèlent contre-productives sur nos formules. Cela nous conduit en définitive à proposer notre propre algorithme de résolution approximée, largement inspiré de deux techniques existantes mais permettant un compromis entre leurs points forts respectifs : de rapide et imprécis à lent mais précis. Nous comparons expérimentalement ces diverses techniques et montrons dans des études de cas le genre d'application que cette approche quantitative nous ouvre, en particulier les fautes matérielles, où un attaquant n'a presque jamais d'opportunité exploitable 100% du temps.

L'atteignabilité robuste et la robustesse quantitative donnent une idée raisonnablement précise de la réplicabilité d'un bug, c'est-à-dire de la capacité d'un attaquant de le déclencher. Si ce n'est ni une condition nécessaire ni suffisante pour que le bug en question soit une vulnérabilité, il s'agit malgré tout d'une information cruciale pour l'évaluation de son impact en termes de sécurité.

Bibliography

[1] Rajeev Alur, Thomas A. Henzinger, and Orna Kupferman. "Alternating-Time Temporal Logic". In: *Journal of the ACM* 49.5 (Sept. 2002), pp. 672–713. ISSN: 0004-5411, 1557-735X. DOI: 10/cgwb3h.

[2] June Andronick et al. "Large-Scale Formal Verification in Practice: A Process Perspective". In: *2012 34th International Conference on Software Engineering (ICSE)*. 2012 34th International Conference on Software Engineering (ICSE). June 2012, pp. 1002–1011. DOI: 10/gphnkb.

[3] Gilles Audemard and Laurent Simon. "On the Glucose SAT Solver". In: *International Journal on Artificial Intelligence Tools* 27.01 (Feb. 2018), p. 1840001. ISSN: 0218-2130. DOI: 10.1142/S0218213018400018.

[4] Thanassis Avgerinos et al. "Automatic Exploit Generation". In: *Communications of the ACM* 57.2 (Feb. 1, 2014), pp. 74–84. ISSN: 0001-0782. DOI: 10.1145/2560217.2560219.

[5] Adnan Aziz et al. "Verifying Continuous Time Markov Chains". In: *Computer Aided Verification*. Ed. by Rajeev Alur and Thomas A. Henzinger. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 1996, pp. 269–276. ISBN: 978-3-540-68599-9. DOI: 10.1007/3-540-61474-5_75.

[6] Rehan Abdul Aziz et al. "# SAT: Projected Model Counting". In: *Theory and Applications of Satisfiability Testing – SAT 2015*. Ed. by Marijn Heule and Sean Weaver. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2015, pp. 121–137. ISBN: 978-3-319-24318-4. DOI: 10/gh6pzs.

[7] Gogul Balakrishnan and Thomas Reps. "WYSINWYX: What You See Is Not What You eXecute". In: *ACM Transactions on Programming Languages and Systems* 32.6 (Aug. 2010), pp. 1–84. ISSN: 0164-0925, 1558-4593. DOI: 10.1145/1749608.1749612.

[8] Roberto Baldoni et al. "A Survey of Symbolic Execution Techniques". In: *ACM Computing Surveys* 51.3 (July 16, 2018), pp. 1–39. ISSN: 0360-0300, 1557-7341. DOI: 10.1145/3182657.

[9] Clark W. Barret et al. "Satisfiability Modulo Theories". In: *Handbook of Satisfiability*. IOS Press. 2009, pp. 825–885. ISBN: 978-1-58603-929-5.

[10] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. *The Satisfiability modulo Theories Library (SMT-LIB)*. www.SMT-LIB.org. 2016.

[11] Clark Barrett et al. "CVC4". In: *Computer Aided Verification*. Ed. by Ganesh Gopalakrishnan and Shaz Qadeer. Vol. 6806. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 171–177. ISBN: 978-3-642-22109-5 978-3-642-22110-1. DOI: 10.1007/978-3-642-22110-1_14.

[12] G. Barthe, P.R. D'Argenio, and T. Rezk. "Secure Information Flow by Self-Composition". In: *Proceedings. 17th IEEE Computer Security Foundations Workshop, 2004.* Proceedings. 17th IEEE Computer Security Foundations Workshop, 2004. June 2004, pp. 100–114. DOI: 10.1109/CSFW.2004.1310735.

[13] Patrick Behm et al. "MéTéor: A Successful Application of B in a Large Project". In: *Proceedings of the Wold Congress on Formal Methods in the Development of Computing Systems-Volume I - Volume I.* FM '99. Berlin, Heidelberg: Springer-Verlag, 1999, pp. 369–387. ISBN: 3-540-66587-0.

[14] Armin Biere, Keijo Heljanko, and Siert Wieringa. *AIGER 1.9 and Beyond.* 11/2. Altenbergerstr. 69, 4040 Linz, Austria: Institute for Formal Models and Verification, Johannes Kepler University, July 2011, 2011.

[15] Armin Biere et al. "CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling Entering the SAT Competition 2020". In: *Proc. of SAT Competition 2020 – Solver and Benchmark Descriptions.* Ed. by Tomas Balyo et al. Vol. B-2020-1. Department of Computer Science Report Series B. University of Helsinki, 2020, pp. 51–53.

[16] Bruno Blanchet et al. "Design and Implementation of a Special-Purpose Static Program Analyzer for Safety-Critical Real-Time Embedded Software". In: *The Essence of Computation: Complexity, Analysis, Transformation.* Ed. by Torben Æ. Mogensen, David A. Schmidt, and I. Hal Sudborough. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2002, pp. 85–108. ISBN: 978-3-540-36377-4. DOI: 10.1007/3-540-36377-7_5.

[17] Roderick Bloem et al., eds. *Handbook of Model Checking.* 1st ed. 2018. Cham: Springer International Publishing : Imprint: Springer, 2018. 1 p. ISBN: 978-3-319-10575-8. DOI: 10.1007/978-3-319-10575-8.

[18] Mateus Borges et al. "Model-Counting Approaches for Nonlinear Numerical Constraints". In: *NASA Formal Methods.* Ed. by Clark Barrett, Misty Davies, and Temesghen Kahsai. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2017, pp. 131–138. ISBN: 978-3-319-57288-8. DOI: 10/ghtsvm.

[19] Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. "What's Decidable About Arrays?" In: *Verification, Model Checking, and Abstract Interpretation.* Ed. by E. Allen Emerson and Kedar S. Namjoshi. Red. by David Hutchison et al. Vol. 3855. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 427–442. ISBN: 978-3-540-31139-3 978-3-540-31622-0. DOI: 10.1007/11609773_28.

[20] Angelo Brillout et al. "Beyond Quantifier-Free Interpolation in Extensions of Presburger Arithmetic". In: *Verification, Model Checking, and Abstract Interpretation.* Ed. by Ranjit Jhala and David Schmidt. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2011, pp. 88–102. ISBN: 978-3-642-18275-4. DOI: 10.1007/978-3-642-18275-4_8.

[21] *Bug 1150468 – VUL-1: DISPUTED: CVE-2019-16230: Kernel-Source: NULL Pointer Dereference in Alloc_workqueue in Drivers/Gpu/Drm/Radeon/Radeon_display.c.* URL: https://bugzilla.suse.com/show_bug.cgi?id=1150468.

[22] Cristian Cadar and Koushik Sen. "Symbolic Execution for Software Testing: Three Decades Later". In: *Communications of the ACM* 56.2 (Feb. 1, 2013), pp. 82–90. ISSN: 0001-0782. DOI: 10.1145/2408776.2408795.

[23] Cristiano Calcagno et al. “Moving Fast with Software Verification”. In: *NASA Formal Methods*. Ed. by Klaus Havelund, Gerard Holzmann, and Rajeev Joshi. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2015, pp. 3–11. ISBN: 978-3-319-17524-9. DOI: 10.1007/978-3-319-17524-9_1.

[24] Sang Kil Cha et al. “Unleashing Mayhem on Binary Code”. In: *2012 IEEE Symposium on Security and Privacy*. 2012 IEEE Symposium on Security and Privacy. May 2012, pp. 380–394. DOI: 10.1109/SP.2012.31.

[25] Supratik Chakraborty et al. “Approximate Probabilistic Inference via Word-Level Counting”. In: *Proceedings of the AAAI Conference on Artificial Intelligence* 30.1 (1 Mar. 5, 2016). ISSN: 2374-3468. URL: https://ojs.aaai.org/index.php/AAAI/article/view/10416.

[26] Mark Chavira and Adnan Darwiche. “On Probabilistic Inference by Weighted Model Counting”. In: *Artificial Intelligence* 172.6-7 (Apr. 1, 2008), pp. 772–799. ISSN: 0004-3702. DOI: 10.1016/j.artint.2007.11.002.

[27] Mark Chavira, Adnan Darwiche, and Manfred Jaeger. “Compiling Relational Bayesian Networks for Exact Inference”. In: *International Journal of Approximate Reasoning*. 2004, pp. 49–56. DOI: 10.1016/j.ijar.2005.10.001.

[28] A. Cimatti et al. “NuSMV: A New Symbolic Model Verifier”. In: *Proceedings Eleventh Conference on Computer-Aided Verification (CAV'99)*. Ed. by N. Halbwachs and D. Peled. Lecture Notes in Computer Science. Trento, Italy: Springer, July 1999, pp. 495–499.

[29] Edmund Clarke, Daniel Kroening, and Flavio Lerda. “A Tool for Checking ANSI-C Programs”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Kurt Jensen and Andreas Podelski. Red. by Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen. Vol. 2988. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 168–176. ISBN: 978-3-540-21299-7 978-3-540-24730-2. DOI: 10.1007/978-3-540-24730-2_15.

[30] Edmund M. Clarke and E. Allen Emerson. “Design and Synthesis of Synchronization Skeletons Using Branching Time Temporal Logic”. In: *Logics of Programs*. Ed. by Dexter Kozen. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 1982, pp. 52–71. ISBN: 978-3-540-39047-3. DOI: 10.1007/BFb0025774.

[31] Edmund M. Clarke and Jeannette M. Wing. “Formal Methods: State of the Art and Future Directions”. In: *ACM Computing Surveys* 28.4 (Dec. 1996), pp. 626–643. ISSN: 0360-0300, 1557-7341. DOI: 10.1145/242223.242257.

[32] Michael R. Clarkson and Fred B. Schneider. “Hyperproperties”. In: *Journal of Computer Security* 18.6 (Sept. 20, 2010). Ed. by Andrei Sabelfeld, pp. 1157–1210. ISSN: 18758924, 0926227X. DOI: 10.3233/JCS-2009-0393.

[33] Michael R. Clarkson et al. “Temporal Logics for Hyperproperties”. In: *Principles of Security and Trust*. Ed. by Martín Abadi and Steve Kremer. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2014, pp. 265–284. ISBN: 978-3-642-54792-8. DOI: 10.1007/978-3-642-54792-8_15.

[34] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. “Terminator: Beyond Safety”. In: *Computer Aided Verification*. Ed. by Thomas Ball and Robert B. Jones. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2006, pp. 415–418. ISBN: 978-3-540-37411-4. DOI: 10.1007/11817963_37.

[35] Patrick Cousot. *Principles of Abstract Interpretation*. Cambridge, MA, USA: MIT Press, Sept. 21, 2021. 832 pp. ISBN: 978-0-262-04490-5.

[36] Crispin Cowan et al. “StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks”. In: 7th {USENIX} Security Symposium ({USENIX} Security 98). 1998. URL: https://www.usenix.org/conference/7th-usenix-security-symposium/stackguard-automatic-adaptive-detection-and-prevention.

[37] Cas Cremers et al. “A Comprehensive Symbolic Analysis of TLS 1.3”. In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. CCS ’17. New York, NY, USA: Association for Computing Machinery, Oct. 30, 2017, pp. 1773–1788. ISBN: 978-1-4503-4946-8. DOI: 10/gf83z9.

[38] *CVE-2014-0160*. URL: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0160.

[39] *Cyber Attacks Hit Two French Hospitals in One Week*. France 24. Feb. 16, 2021. URL: https://www.france24.com/en/europe/20210216-cyber-attacks-hit-two-french-hospitals-in-one-week.

[40] Lesly-Ann Daniel, Sebastien Bardin, and Tamara Rezk. “Binsec/Rel: Efficient Relational Symbolic Execution for Constant-Time at Binary-Level”. In: *2020 IEEE Symposium on Security and Privacy (SP)*. 2020 IEEE Symposium on Security and Privacy (SP). San Francisco, CA, USA: IEEE, May 2020, pp. 1021–1038. ISBN: 978-1-72813-497-0. DOI: 10.1109/SP40000.2020.00074.

[41] Adnan Darwiche. “Decomposable Negation Normal Form”. In: *Journal of the ACM* 48.4 (July 1, 2001), pp. 608–647. ISSN: 0004-5411. DOI: 10/czk9nk.

[42] Adnan Darwiche. “New Advances in Compiling CNF to Decomposable Negation Normal Form”. In: *Proceedings of the 16th European Conference on Artificial Intelligence*. ECAI’04. NLD: IOS Press, Aug. 22, 2004, pp. 318–322. ISBN: 978-1-58603-452-8.

[43] Adnan Darwiche. “On the Tractable Counting of Theory Models and Its Application to Truth Maintenance and Belief Revision”. In: *Journal of Applied Non-Classical Logics* 11 (2000), pp. 1–2.

[44] Robin David et al. “BINSEC/SE: A Dynamic Symbolic Execution Toolkit for Binary-Level Analysis”. In: *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. 2016 IEEE 23rd International Conference on Software Analysis, Evolution and Reengineering (SANER). Suita: IEEE, Mar. 2016, pp. 653–656. ISBN: 978-1-5090-1855-0. DOI: 10.1109/SANER.2016.43.

[45] Robin David et al. “Specification of Concretization and Symbolization Policies in Symbolic Execution”. In: *Proceedings of the 25th International Symposium on Software Testing and Analysis*. ISSTA 2016. New York, NY, USA: Association for Computing Machinery, July 18, 2016, pp. 36–46. ISBN: 978-1-4503-4390-9. DOI: 10.1145/2931037.2931048.

[46] Jesús A. De Loera et al. “Effective Lattice Point Counting in Rational Convex Polytopes”. In: *Journal of Symbolic Computation*. Symbolic Computation in Algebra and Geometry 38.4 (Oct. 1, 2004), pp. 1273–1302. ISSN: 0747-7171. DOI: 10/cf2mq7.

[47] Leonardo de Moura and Nikolaj Bjørner. “Efficient E-Matching for SMT Solvers”. In: *Automated Deduction – CADE-21*. Ed. by Frank Pfenning. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2007, pp. 183–198. ISBN: 978-3-540-73595-3. DOI: 10.1007/978-3-540-73595-3_13.

[48] Leonardo de Moura and Nikolaj Bjørner. “Z3: An Efficient SMT Solver”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by C. R. Ramakrishnan and Jakob Rehof. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2008, pp. 337–340. ISBN: 978-3-540-78800-3. DOI: 10.1007/978-3-540-78800-3_24.

[49] Adel Djoudi and Sébastien Bardin. "BINSEC: Binary Code Analysis with Low-Level Regions". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Christel Baier and Cesare Tinelli. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2015, pp. 212–217. ISBN: 978-3-662-46681-0. DOI: 10.1007/978-3-662-46681-0_17.

[50] Eelco Dolstra. "The Purely Functional Software Deployment Model". University of Utrecht, 2006. ISBN: 9789039341308.

[51] Arnaud Durand, Miki Hermann, and Phokion G. Kolaitis. "Subtractive Reductions and Complete Problems for Counting Complexity Classes". In: *Mathematical Foundations of Computer Science 2000*. International Symposium on Mathematical Foundations of Computer Science. Springer, Berlin, Heidelberg, Aug. 28, 2000, pp. 323–332. DOI: 10/b5hzzb.

[52] Louis Dureuil et al. "FISSC: A Fault Injection and Simulation Secure Collection". In: *Computer Safety, Reliability, and Security*. Ed. by Amund Skavhaug, Jérémie Guiochet, and Friedemann Bitsch. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2016, pp. 3–11. ISBN: 978-3-319-45477-1. DOI: 10/ggskcw.

[53] Hélène Fargier and Pierre Marquis. "On the Use of Partially Ordered Decision Graphs in Knowledge Compilation and Quantified Boolean Formulae". In: *Proceedings, the Twenty-First National Conference on Artificial Intelligence and the Eighteenth Innovative Applications of Artificial Intelligence Conference, July 16-20, 2006, Boston, Massachusetts, USA*. AAAI Press, 2006, pp. 42–47. URL: http://www.aaai.org/Library/AAAI/2006/aaai06-007.php.

[54] Benjamin Farinier. "Decision Procedures for Vulnerability Analysis". Université Grenoble-Alpes, 2020.

[55] Benjamin Farinier et al. "Arrays Made Simpler: An Efficient, Scalable and Thorough Preprocessing". In: LPAR-22. 22nd International Conference on Logic for Programming, Artificial Intelligence and Reasoning. Oct. 2018, pp. 363–344. DOI: 10.29007/dc9b.

[56] Benjamin Farinier et al. "Model Generation for Quantified Formulas: A Taint-Based Approach". In: *Computer Aided Verification*. Ed. by Hana Chockler and Georg Weissenbacher. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2018, pp. 294–313. ISBN: 978-3-319-96142-2. DOI: 10.1007/978-3-319-96142-2_19.

[57] Daniel Fremont, Markus Rabe, and Sanjit Seshia. "Maximum Model Counting". In: *Proceedings of the AAAI Conference on Artificial Intelligence* 31.1 (1 Feb. 12, 2017). ISSN: 2374-3468. URL: https://ojs.aaai.org/index.php/AAAI/article/view/11138.

[58] Yeting Ge and Leonardo de Moura. "Complete Instantiation for Quantified Formulas in Satisfiabiliby Modulo Theories". In: *Computer Aided Verification*. Ed. by Ahmed Bouajjani and Oded Maler. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2009, pp. 306–320. ISBN: 978-3-642-02658-4. DOI: 10.1007/978-3-642-02658-4_25.

[59] Jaco Geldenhuys, Matthew B. Dwyer, and Willem Visser. "Probabilistic Symbolic Execution". In: *Proceedings of the 2012 International Symposium on Software Testing and Analysis*. ISSTA 2012. New York, NY, USA: Association for Computing Machinery, July 15, 2012, pp. 166–176. ISBN: 978-1-4503-1454-1. DOI: 10/ggbn25.

[60] Christophe Giraud and Hugues Thiebeauld. "A Survey on Fault Attacks". In: *Smart Card Research and Advanced Applications VI*. Ed. by Jean-Jacques Quisquater et al. IFIP International Federation for Information Processing. Boston, MA: Springer US, 2004, pp. 159–176. ISBN: 978-1-4020-8147-7. DOI: 10/b5jk83.

[61] Guillaume Girol, Benjamin Farinier, and Sébastien Bardin. "Not All Bugs Are Created Equal, But Robust Reachability Can Tell the Difference". In: *Computer Aided Verification*. Ed. by Alexandra Silva and K. Rustan M. Leino. Lecture Notes in Computer Science. Cham: Springer, 2021, pp. 669–693. ISBN: 978-3-030-81685-8. DOI: 10/gmn5z6.

[62] Patrice Godefroid. "Higher-Order Test Generation". In: *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '11. New York, NY, USA: Association for Computing Machinery, June 4, 2011, pp. 258–269. ISBN: 978-1-4503-0663-8. DOI: 10/dc3cxv.

[63] Patrice Godefroid, Nils Klarlund, and Koushik Sen. "DART: Directed Automated Random Testing". In: *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '05. Chicago, IL, USA: Association for Computing Machinery, June 12, 2005, pp. 213–223. ISBN: 978-1-59593-056-9. DOI: 10.1145/1065010.1065036.

[64] Patrice Godefroid, Michael Y. Levin, and David Molnar. "SAGE: Whitebox Fuzzing for Security Testing: SAGE Has Had a Remarkable Impact at Microsoft." In: *Queue* 10.1 (Jan. 11, 2012), pp. 20–27. ISSN: 1542-7730. DOI: 10.1145/2090147.2094081.

[65] J. A. Goguen and J. Meseguer. "Security Policies and Security Models". In: *1982 IEEE Symposium on Security and Privacy*. 1982 IEEE Symposium on Security and Privacy. Oakland, CA, USA: IEEE, Apr. 1982, pp. 11–11. ISBN: 978-0-8186-0410-2. DOI: 10.1109/SP.1982.10014.

[66] Carla Gomes and Meinolf Sellmann. "Streamlined Constraint Reasoning". In: *Principles and Practice of Constraint Programming – CP 2004*. Ed. by Mark Wallace. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2004, pp. 274–289. ISBN: 978-3-540-30201-8. DOI: 10/ft6grr.

[67] Carla P. Gomes, Ashish Sabharwal, and Bart Selman. "Model Counting". In: *Handbook of Satisfiability*. IOS Press, 2008.

[68] Reiner Hähnle and Marieke Huisman. "Deductive Software Verification: From Pen-and-Paper Proofs to Industrial Tools". In: *Computing and Software Science: State of the Art and Perspectives*. Ed. by Bernhard Steffen and Gerhard Woeginger. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2019, pp. 345–373. ISBN: 978-3-319-91908-9. DOI: 10.1007/978-3-319-91908-9_18.

[69] Trevor Hansen, Peter Schachte, and Harald Søndergaard. "State Joining and Splitting for the Symbolic Execution of Binaries". In: *Runtime Verification*. Ed. by Saddek Bensalem and Doron A. Peled. Vol. 5779. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 76–92. ISBN: 978-3-642-04693-3 978-3-642-04694-0. DOI: 10.1007/978-3-642-04694-0_6.

[70] Hans Hansson and Bengt Jonsson. "A Logic for Reasoning about Time and Reliability". In: *Formal Aspects of Computing* 6.5 (Sept. 1994), pp. 512–535. ISSN: 0934-5043, 1433-299X. DOI: 10.1007/BF01211866.

[71] Sergiu Hart, Micha Sharir, and Amir Pnueli. "Termination of Probabilistic Concurrent Program". In: *ACM Transactions on Programming Languages and Systems* 5.3 (July 1, 1983), pp. 356–380. ISSN: 0164-0925. DOI: 10.1145/2166.357214.

[72] Sean Heelan. "Automatic Generation of Control Flow Hijacking Exploits for Software Vulnerabilities". MA thesis. University of Oxford, 2009.

[73] Jonathan Heusser and Pasquale Malacaria. “Quantifying Information Leaks in Software”. In: *Proceedings of the 26th Annual Computer Security Applications Conference on - ACSAC ’10*. The 26th Annual Computer Security Applications Conference. Austin, Texas: ACM Press, 2010, p. 261. ISBN: 978-1-4503-0133-6. DOI: 10.1145/1920261.1920300.

[74] Christian Holler, Kim Herzig, and Andreas Zeller. “Fuzzing with Code Fragments”. In: *21st USENIX Security Symposium (USENIX Security 12)*. Bellevue, WA: USENIX Association, Aug. 2012, pp. 445–458. ISBN: 978-931971-95-9. URL: https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/holler.

[75] Jinbo Huang. “Combining Knowledge Compilation and Search for Conformant Probabilistic Planning”. In: *Proceedings of the Sixteenth International Conference on International Conference on Automated Planning and Scheduling*. ICAPS’06. Cumbria, UK: AAAI Press, June 6, 2006, pp. 253–262. ISBN: 978-1-57735-270-9.

[76] Jinbo Huang, M. Chavira, and Adnan Darwiche. “Solving MAP Exactly by Searching on Compiled Arithmetic Circuits”. In: *AAAI*. 2006.

[77] Seonmo Kim and Stephen McCamant. “Bit-Vector Model Counting Using Statistical Estimation”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Dirk Beyer and Marieke Huisman. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2018, pp. 133–151. ISBN: 978-3-319-89960-2. DOI: 10/ghtr84.

[78] Greg Kroah-Hartman. *Re: Memory Leak in U_audio_start_playback [LWN.Net]*. URL: https://lwn.net/ml/linux-kernel/20201027164343.GA1523116@kroah.com/.

[79] Damian Kurpiewski, Michał Knapik, and Wojciech Jamroga. “On Domination and Control in Strategic Ability”. In: *AAMAS* (2019), p. 9.

[80] Volodymyr Kuznetsov et al. “Efficient State Merging in Symbolic Execution”. In: *ACM SIGPLAN Notices* 47.6 (June 11, 2012), pp. 193–204. ISSN: 0362-1340. DOI: 10.1145/2345156.2254088.

[81] Jean-Marie Lagniez and Pierre Marquis. “A Recursive Algorithm for Projected Model Counting”. In: *Proceedings of the AAAI Conference on Artificial Intelligence* 33 (July 17, 2019), pp. 1536–1543. ISSN: 2374-3468, 2159-5399. DOI: 10/ghkjdq.

[82] Jean-Marie Lagniez and Pierre Marquis. “An Improved Decision-DNNF Compiler”. In: *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence*. Twenty-Sixth International Joint Conference on Artificial Intelligence. Melbourne, Australia: International Joint Conferences on Artificial Intelligence Organization, Aug. 2017, pp. 667–673. ISBN: 978-0-9992411-0-3. DOI: 10/gh6rkj.

[83] Johannes Lampel et al. “When Life Gives You Oranges: Detecting and Diagnosing Intermittent Job Failures at Mozilla”. In: *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2021. New York, NY, USA: Association for Computing Machinery, Aug. 20, 2021, pp. 1381–1392. ISBN: 978-1-4503-8562-6. DOI: 10.1145/3468264.3473931.

[84] Nian-Ze Lee and Jie-Hong R. Jiang. “Towards Formal Evaluation and Verification of Probabilistic Design”. In: *Proceedings of the 2014 IEEE/ACM International Conference on Computer-Aided Design*. ICCAD ’14. San Jose, California: IEEE Press, Nov. 3, 2014, pp. 340–347. ISBN: 978-1-4799-6277-8.

[85] Nian-Ze Lee, Yen-Shi Wang, and Jie-Hong R. Jiang. "Solving Exist-Random Quantified Stochastic Boolean Satisfiability via Clause Selection". In: *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence*. Twenty-Seventh International Joint Conference on Artificial Intelligence {IJCAI-18}. Stockholm, Sweden: International Joint Conferences on Artificial Intelligence Organization, July 2018, pp. 1339–1345. ISBN: 978-0-9992411-2-7. DOI: 10.24963/ijcai.2018/186.

[86] Xavier Leroy et al. "CompCert - A Formally Verified Optimizing Compiler". In: ERTS 2016: Embedded Real Time Software and Systems, 8th European Congress. Jan. 27, 2016. URL: https://hal.inria.fr/hal-01238879.

[87] M. L. Littman, J. Goldsmith, and M. Mundhenk. "The Computational Complexity of Probabilistic Planning". In: *Journal of Artificial Intelligence Research* 9 (Aug. 1, 1998), pp. 1–36. ISSN: 1076-9757. DOI: 10.1613/jair.505.

[88] Michael L. Littman. "Probabilistic Propositional Planning: Representations and Complexity". In: *In Proceedings of the Fourteenth National Conference on Artificial Intelligence*. MIT Press, 1997, pp. 748–754.

[89] Michael L. Littman, Stephen M. Majercik, and Toniann Pitassi. "Stochastic Boolean Satisfiability". In: *Journal of Automated Reasoning* 27.3 (3 Oct. 1, 2001), pp. 251–296. ISSN: 1573-0670. DOI: 10.1023/A:1017584715408.

[90] Benjamin Livshits et al. "In Defense of Soundiness: A Manifesto". In: *Communications of the ACM* 58.2 (Jan. 28, 2015), pp. 44–46. ISSN: 0001-0782. DOI: 10/gpbgzs.

[91] Qingzhou Luo et al. "An Empirical Analysis of Flaky Tests". In: *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. FSE 2014. New York, NY, USA: Association for Computing Machinery, Nov. 11, 2014, pp. 643–653. ISBN: 978-1-4503-3056-5. DOI: 10.1145/2635868.2635920.

[92] Stephen M. Majercik and Byron Boots. "DC-SSAT: A Divide-and-Conquer Approach to Solving Stochastic Satisfiability Problems Efficiently". In: *Proceedings of the 20th National Conference on Artificial Intelligence - Volume 1*. AAAI'05. Pittsburgh, Pennsylvania: AAAI Press, July 9, 2005, pp. 416–422. ISBN: 978-1-57735-236-5.

[93] Barton P. Miller et al. *Fuzz Revisited: A Re-Examination of the Reliability of UNIX Utilities and Services*. University of Wisconsin-Madison Department of Computer Sciences, 1995.

[94] Christian Muise et al. "Dsharp: Fast d-DNNF Compilation with sharpSAT". In: *Advances in Artificial Intelligence*. Ed. by Leila Kosseim and Diana Inkpen. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2012, pp. 356–361. ISBN: 978-3-642-30353-1. DOI: 10/gjjsfh.

[95] Nicholas Nethercote and Julian Seward. "Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation". In: *ACM Sigplan notices* 42.6 (2007), pp. 89–100. DOI: 10/dw6n36.

[96] Aina Niemetz and Mathias Preiner. *Bitwuzla at the SMT-COMP 2020*. May 29, 2020. arXiv: 2006.01621 [cs]. URL: http://arxiv.org/abs/2006.01621.

[97] Aina Niemetz, Mathias Preiner, and Armin Biere. "Boolector 2.0: System Description". In: *Journal on Satisfiability, Boolean Modeling and Computation* 9.1 (June 1, 2015), pp. 53–58. ISSN: 15740617. DOI: 10/ghv4cd.

[98] Peter W. O'Hearn. "Incorrectness Logic". In: *Proceedings of the ACM on Programming Languages* 4 (POPL Jan. 2020), pp. 1–32. ISSN: 2475-1421, 2475-1421. DOI: 10.1145/3371078.

[99] Christos H. Papadimitriou. "Games against Nature". In: *Journal of Computer and System Sciences* 31.2 (Oct. 1, 1985), pp. 288–301. ISSN: 0022-0000. DOI: 10.1016/0022-0000(85)90045-5.

[100] J. D. Park and A. Darwiche. "Complexity Results and Approximation Strategies for MAP Explanations". In: *Journal of Artificial Intelligence Research* 21 (2004), pp. 101–133. ISSN: 1076-9757. DOI: 10.1613/jair.1236.

[101] Knot Pipatsrisawat and Adnan Darwiche. "A New D-DNNF-Based Bound Computation Algorithm for Functional E-MAJSAT". In: *IJCAI*. 2009.

[102] David A. Ramos and Dawson Engler. "Under-Constrained Symbolic Execution: Correctness Checking for Real Code". In: *Proceedings of the 24th USENIX Conference on Security Symposium*. SEC'15. USA: USENIX Association, Aug. 12, 2015, pp. 49–64. ISBN: 978-1-931971-23-2.

[103] Frederic Recoules et al. "Get Rid of Inline Assembly through Verification-Oriented Lifting". In: *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE). San Diego, CA, USA: IEEE, Nov. 2019, pp. 577–589. ISBN: 978-1-72812-508-4. DOI: 10.1109/ASE.2019.00060.

[104] Andrew Reynolds et al. "Finite Model Finding in SMT". In: *Computer Aided Verification*. Ed. by Natasha Sharygina and Helmut Veith. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2013, pp. 640–655. ISBN: 978-3-642-39799-8. DOI: 10.1007/978-3-642-39799-8_42.

[105] H. G. Rice. "Classes of Recursively Enumerable Sets and Their Decision Problems". In: *Transactions of the American Mathematical Society* 74.2 (1953), pp. 358–366. ISSN: 0002-9947, 1088-6850. DOI: 10.1090/S0002-9947-1953-0053041-6.

[106] Xavier Rival and Kwangkeun Yi. *Introduction to Static Analysis: An Abstract Interpretation Perspective*. Cambridge, MA, USA: MIT Press, Feb. 11, 2020. 320 pp. ISBN: 978-0-262-04341-0.

[107] Neil Robertson and P. D Seymour. "Graph Minors. II. Algorithmic Aspects of Tree-Width". In: *Journal of Algorithms* 7.3 (Sept. 1, 1986), pp. 309–322. ISSN: 0196-6774. DOI: 10.1016/0196-6774(86)90023-4.

[108] Dan Roth. "On the Hardness of Approximate Reasoning". In: *Artificial Intelligence* 82.1 (Apr. 1, 1996), pp. 273–302. ISSN: 0004-3702. DOI: 10.1016/0004-3702(94)00092-1.

[109] 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*. ESEC/FSE-13. New York, NY, USA: Association for Computing Machinery, Sept. 1, 2005, pp. 263–272. ISBN: 978-1-59593-014-9. DOI: 10.1145/1081706.1081750.

[110] Konstantin Serebryany et al. "AddressSanitizer: A Fast Address Sanity Checker". In: 2012 USENIX Annual Technical Conference (USENIX ATC 12). 2012, pp. 309–318. URL: https://www.usenix.org/conference/atc12/technical-sessions/presentation/serebryany.

[111] Kostya Serebryany. "OSS-Fuzz - Google's Continuous Fuzzing Service for Open Source Software". In: (2017). URL: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/serebryany.

[112] Shubham Sharma et al. “GANAK: A Scalable Probabilistic Exact Model Counter”. In: *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI-19*. International Joint Conferences on Artificial Intelligence Organization, July 2019, pp. 1169–1176. DOI: 10/gh5qx7.

[113] Solomon Eyal Shimony. “Finding MAPs for Belief Networks Is NP-hard”. In: *Artificial Intelligence* 68.2 (Aug. 1, 1994), pp. 399–410. ISSN: 0004-3702. DOI: 10.1016/0004-3702(94)90072-8.

[114] Yan Shoshitaishvili et al. “SOK: (State of) The Art of War: Offensive Techniques in Binary Analysis”. In: *2016 IEEE Symposium on Security and Privacy (SP)*. 2016 IEEE Symposium on Security and Privacy (SP). May 2016, pp. 138–157. DOI: 10.1109/SP.2016.17.

[115] Julien Signoles, Nikolai Kosmatov, and Kostyantyn Vorobyov. “E-ACSL, a Runtime Verification Tool for Safety and Security of C Programs (Tool Paper).” In: *RV-CuBES*. 2017, pp. 164–173.

[116] Fu Song and Tayssir Touili. “Efficient CTL Model-Checking for Pushdown Systems”. In: *Theoretical Computer Science* 549 (Sept. 11, 2014), pp. 127–145. ISSN: 0304-3975. DOI: 10.1016/j.tcs.2014.07.001.

[117] Marc Thurley. “sharpSAT – Counting Models with Advanced Component Caching and Implicit BCP”. In: *Theory and Applications of Satisfiability Testing - SAT 2006*. Ed. by Armin Biere and Carla P. Gomes. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2006, pp. 424–429. ISBN: 978-3-540-37207-3. DOI: 10/dxdj4v.

[118] S. Toda. “On the Computational Power of PP and (+)P”. In: *Proceedings of the 30th Annual Symposium on Foundations of Computer Science*. SFCS ’89. USA: IEEE Computer Society, Oct. 30, 1989, pp. 514–519. ISBN: 978-0-8186-1982-3. DOI: 10/frvmnp.

[119] L. G. Valiant. “The Complexity of Computing the Permanent”. In: *Theoretical Computer Science* 8.2 (Jan. 1, 1979), pp. 189–201. ISSN: 0304-3975. DOI: 10/c6rvdw.

[120] Anjiang Wei et al. “Preempting Flaky Tests via Non-Idempotent-Outcome Tests”. In: *Proceedings of the 44th International Conference on Software Engineering*. ICSE ’22. New York, NY, USA: Association for Computing Machinery, May 21, 2022, pp. 1730–1742. ISBN: 978-1-4503-9221-1. DOI: 10.1145/3510003.3510170.

[121] Nicky Williams et al. “PathCrawler: Automatic Generation of Path Tests by Combining Static and Dynamic Analysis”. In: *Dependable Computing - EDCC 5*. Ed. by Mario Dal Cin, Mohamed Kaâniche, and András Pataricza. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2005, pp. 281–292. ISBN: 978-3-540-32019-7. DOI: 10.1007/11408901_21.

[122] Yexiang Xue et al. “Solving Marginal MAP Problems with NP Oracles and Parity Constraints”. In: *Advances in Neural Information Processing Systems*. Vol. 29. Curran Associates, Inc., 2016. URL: https://proceedings.neurips.cc/paper/2016/hash/a532400ed62e772b9dc0b86f46e583ff-Abstract.html.