Software security : combining fuzzing and symbolic methods for vulnerability detection

Yaëlle Vinçont

université
PARIS-SACLAY

Software Security: Combining Fuzzing and Symbolic Methods for Vulnerability Detection

Fuzzing et méthodes symboliques pour la détection de vulnérabilités à large échelle

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 des universités, et le co-encadrement de **Sébastien BARDIN**, chercheur.

Thèse soutenue à Paris-Saclay, le 15 décembre 2021, par

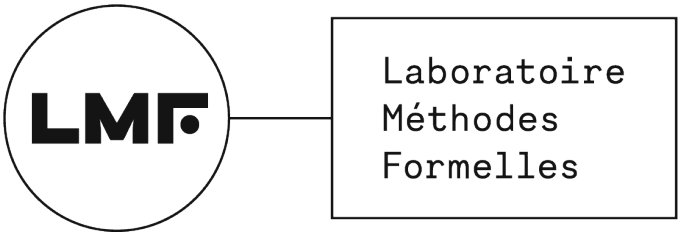
Yaëlle VINÇONT

Composition du jury

Mihaela SIGHIREANU Professeure des Universités, Ecole Normale Supérieure Paris-Saclay	Présidente
Jean-Yves MARION Professeur des Universités, Ecole Nationale Supérieure des Mines de Nancy	Rapporteur & Examinateur
Michail PAPADAKIS Maître de Conférences, HDR, Université du Luxembourg	Rapporteur & Examinateur
Emmanuelle ENCRENAZ Maîtresse de Conférences, Sorbonne Université	Examinatrice
Sylvain CONCHON Professeur des Universités, Université Paris-Saclay	Directeur de thèse

THESE DE DOCTORAT
NNT : 2021UPASG112

Remerciements institutionnels



Cette thèse n'aurait pu se dérouler sans le soutien matériel de deux acteurs.

Je tiens donc à remercier le Commissariat à l'Énergie Atomique et aux Énergies Alternatives (CEA) pour avoir financé les trois premières années de ma thèse, et en particulier l'Institut List pour m'avoir accueillie dans ses locaux durant cette période.

Je remercie également le Laboratoire Méthodes Formelles (LMF) de l'Université Paris-Saclay pour le financement de ma prolongation en quatrième année, et son accueil.

Remerciements

As tradition dictates, I shall start this work (and finish writing it) by thanking all who helped me throughout the process. This first part is in English, but the more personal acknowledgements below will be in French.

First, I want to thank Jean-Yves Marion and Michail Papadakis for agreeing to review my manuscript. Thank you for your feedback and questions.

Thanks also to Mihaela Sighireanu for presiding my defense, and Emmanuelle Encrenaz for being part of the jury. I enjoyed presenting my work to you, as well as the discussions that happened during the question session.

I would like to thank Sylvain Conchon, my academic supervisor. Thank you for accepting to join the adventure midway, and welcoming into your team during those last few months. I enjoyed our scientific discussions, and your advice.

Finally, I want to thank Sébastien Bardin for being my co-supervisor during those four years. Thanks for offering me this opportunity, and everything you did along the way. I learned a lot under your guidance, both scientifically and academically.

Je passe maintenant de la langue de Shakespeare à celle de Molière pour remercier celleux qui m'ont entourée pendant ces quatre ans.

On dit qu'il faut un village pour élever un enfant, je pense qu'il en est de même pour une thèse… De fait, le meilleur moyen de n'oublier personne est de ne nommer personne, mais je suis certaine que vous vous reconnaîtrez.

Merci aux équipes qui m'ont accueillie. Équipes au pluriel, car j'ai eu la chance de commencer ma thèse au LSL (CEA) et de la finir au LMF (Université Paris-Saclay). Merci pour votre accueil, pour les discussions scientifiques et les pauses café.

Merci aux doctorant·e·s qui ont partagé ces quatre années avec moi. De ces années, il me restera toujours les bons moments (les sorties, la Savoie, les post-its…), mais aussi notre entraide dans les moins bons. Mention toute particulière pour mes co-bureaux, officiels et officieux!

Merci à mes ami·e·s. Celleux rencontré·e·s au lycée, sur les bancs de la fac, autour d'un jeu de cartes. Les collègues que je voyais tous les jours, les colocs (presque) tous les soirs, et les "pocket friends", toujours joignables grâce à Internet. Merci pour ces soirées, ces week-ends, ces discussions, qui m'ont permis de me changer les idées et de survivre à la pression de la thèse.

Merci, pour finir, à ma famille, à deux et quatre pattes. Merci de m'avoir supportée (… et soutenue) pendant ces quatre années. Sans forcément comprendre ce que je faisais, vous avez été une oreille attentive, une source de réconfort mais aussi de motivation. Je n'y serais pas arrivée sans vous.

Résumé

Alors que les programmes informatiques se répandent, le risque de bugs augmente. Dans cette thèse, nous voulons trouver d'éventuels bugs dans des programmes finis et publics.

Pour cela, nous utilisons la génération automatique de tests. Complémentant les tests écrits à la main, les générateurs de tests fabriquent automatiquement une série de tests, avec pour but de maximiser la couverture de code et de minimiser l'effort humain. Actuellement, les techniques de génération de test les plus répandues dans l'académique et dans l'industrie sont basées sur l'exécution symbolique ou le fuzzing.

- L'exécution symbolique vise à explorer complètement les chemins d'exécution. Pour cela, chaque chemin est exécuté sur une entrée symbolique, et une contrainte est inférée sur cette entrée. Quand l'analyse atteint la fin d'un chemin, cette contrainte est un prédicat de chemin, et l'exécution concrète du programme sur n'importe laquelle de ses solutions suivra le chemin voulu. De telles entrées concrètes sont générées avec un solveur de contraintes, et forment la série de tests. Cependant, il n'est pas toujours possible d'explorer tous les chemins en un temps raisonnable, et il est souvent nécessaire de borner l'exploration.

- Le fuzzing vise à exécuter le programme sur de nombreuses entrées, en espérant explorer tous les chemins possibles. Il dépend donc d'une génération d'entrées rapide et facile. Tandis que les fuzzers de base fonctionnent en boîte noire et génèrent des entrées aléatoires indépendamment du programme, les fuzzers en boîte grise utilisent une analyse pour obtenir des informations à propos du programme. Ces informations sont alors utilisées pour améliorer la génération d'entrées. Cependant, malgré ces améliorations, les fuzzers ont toujours de la difficulté à trouver la solution de conditions qui ont une faible probabilité d'être vraies.

Ainsi, l'exécution symbolique et le fuzzing exhibent des forces et des faiblesses complémentaires, nous poussant à les combiner.

Pendant cette thèse, nous avons développé une technique de génération de test automatisée, qui combine la puissance de raisonnement de l'exécution symbolique pour s'attaquer au code complexe, et le faible coût du fuzzing pour générer des entrées efficacement.

La solution que nous proposons combine deux nouvelles idées: l'Exécution Symbolique Légère (ESL) et le Fuzzing Contraint. L'Exécution Symbolique Légère est une variante de l'exécution symbolique où l'analyse s'arrête sur une condition d'un chemin, plutôt qu'à la fin, et le langage de contraintes ciblé est réduit à un fragment facilement énumérable de l'habituel. Par conséquent, dériver des prédicats de chemin (corrects) dans ce langage est plus compliqué, mais il est

facile d'énumérer des entrées exerçant un chemin, sans utiliser de solveur de contraintes. Deuxièmement, le Fuzzer Contraint manipule une entrée et une contrainte facilement énumérable, et génère de nouvelles entrées qui satisfont la contrainte et suivent donc le chemin, jusqu'à la condition ciblée. En général, l'ESL guidera l'exploration au-delà des conditions difficiles et vers les parties intéressantes du code, tandis que le fuzzer contraint créera efficacement des entrées, y compris des solutions aux contraintes. Cela nous permet d'explorer le programme sans systématiquement faire appel à l'analyse symbolique, et supprime la dépendance à un solveur pour créer des entrées satisfaisant les contraintes.

Nous avons combiné ces deux technologies au sein de l'outil ConFuzz, qui est intégré à la plateforme d'analyse de code binaire Binsec. L'exécution symbolique légère a été créée pendant cette thèse, en OCaml, et réutilise seulement certains greffons de Binsec pour obtenir et représenter la trace à analyser. Le fuzzeur contraint a été créé en modifiant AFL, qui représente l'état de l'art du fuzzing en boîte grise, écrit en C.

Nous avons ensuite évalué les performances de ConFuzz sur LAVA-M, un banc de test standard du fuzzing. Il est composé d'applications de la librairie binutils, dans lesquels des vulnérabilités ont été injectées automatiquement. Nous comparons le nombre de vulnérabilités détectées par ConFuzz à celles détectées par AFL et Klee, respectivement l'état de l'art du fuzzing et de l'exécution symbolique, ainsi que QSYM, un outil combinant fuzzing et exécution symbolique.

Les résultats montrent que ConFuzz est plus efficace que du fuzzing ou de l'exécution symbolique traditionnels, même quand on additionne les vulnérabilités trouvées par les deux techniques. Quant à QSYM, ConFuzz montre de meilleurs résultats dans deux des trois tests, des résultats très encourageants.

Dans la suite de ce travail, nous voudrions continuer à améliorer la précision des contraintes calculées par l'exécution symbolique légère (tout en conservant la condition d'énumération facile). Une autre piste qu'il pourrait être intéressant d'explorer est de combiner ConFuzz à de l'exécution symbolique classique, pour gérer les cas où notre analyse ne suffit pas à créer une contrainte permettant de passer une condition difficile.

Abstract

As computer programs spread, the risk of bugs increases. In this thesis, we want to find possible bugs in finished and released programs.

We do this through automatic test generation, a major topic in software engineering and security. A complement to hand-crafted tests, test generators automatically build test suites, aiming to maximize program coverage and minimize human effort. Currently, most test generation techniques and tools studied by researchers and applied in industry rely on some form of either symbolic execution or fuzzing.

- Symbolic execution aim to exhaustively explore the possible execution paths. It achieves this by executing each path on a symbolic input, and inferring a constraint on said input. When the analysis reaches the end of a path, this constraint is a path predicate, and a concrete execution of the program on any of its solution will follow the intended path. Such test cases are generated using an off-the-shelf solver, and form the test suite. However, it is not always possible to explore all paths in reasonable time, and we often have to bound the exploration.

- Fuzzing aims to run the program on many test cases, in order to hopefully trigger all possible paths. As such, it relies on quick and easy test case generation. While the most basic fuzzers function in a blackbox manner and generate random test cases independently from the program, greybox fuzzers also rely on an analysis to gain some information about the program. This information is then used to make the test case generation more efficient. However, despite this improvement, fuzzers still struggle with finding the solution to conditions that have a low probability of being true, such as password checks.

Hence, symbolic execution and fuzzing exhibit rather complementary strengths and weaknesses, calling for a proper integration between the two techniques.

During this thesis, we developed an automated test generation technique, combining the reasoning power of symbolic execution to tackle complex code with the light cost of greybox fuzzing to generate test cases efficiently.

The solution we propose combines two novel ideas: Lightweight Symbolic Execution (LSE) and Constrained Fuzzing. Lightweight Symbolic Execution is a variant of symbolic execution where the analysis targets a condition on a path, rather than a full path, and the target constraint language is restricted to an easily-enumerable fragment of the usual one. As a consequence, deriving (correct) path predicates in this language is more complicated but test cases following a given path are then easy to enumerate, without using any off-the-shelf constraint solver. Second, a Constrained Fuzzer operates over a test case and an easily-enumerable constraint in order to quickly generate test cases which

follow the intended path, up to the targeted condition. Overall, LSE will lead the exploration past difficult conditions and towards interesting parts of the code, while the constrained fuzzer will efficiently create test cases, including solutions to the constraints. This allows us to explore the program without systematically relying on symbolic analysis, and removes the need for an SMT solver to create test cases satisfying the constraints.

We evaluated the performances of the resulting tool, called ConFuzz, on a standard fuzzing benchmark, and found that we improved upon the performance of standard fuzzing and symbolic execution.

Contents

Chapter 1

Introduction

Contents

Context

Anywhere there are programs, there is a risk of bugs. And the more programs we run, on computers but also phones, fridges, planes, etc, the more important it is to find and fix said bugs. Especially since the consequences range from inconvenient – e.g crashing the program – to devastating – e.g leaking private information, a reality that is reflected vocabulary-wise by the distinction between bugs and vulnerabilities.

Bugs and Vulnerabilities *Bugs* are defined as any event where the program does not behave the way it is supposed to. They can have many different causes, but they are usually accidentally introduced by the developers. For example, see the program in Figure 1.1: it first creates a table and initialize its content, then it goes through the table and print the data for each cell. There is a mistake though, as the second loop accesses `tab[5]`, which is not actually part of the table.

This is an *undefined behavior* in C, meaning the compiler can do anything. On my computer, compiled with `gcc` and `-O0` optimization, it just prints the uninitialized content of the memory at this address: `32764`. It could have also

```c
#include <stdio.h>

int main (int argc, void* argv[]) {
  int tab[5];
  int cnt;

  for (cnt = 0; cnt < 5; cnt++)
    tab[cnt] = cnt+1;

  for (cnt = 0; cnt <= 5; cnt++)
    printf("%i-", tab[cnt]);
}
```

Figure 1.1: Example of out-of-bounds array access

crashed the program with a "Segmentation Fault" error message, if we were accessing memory we do not have the rights to. Alternatively, if this code was running on a server, and a malicious client sent a request for a number of bytes greater than the ones they had initialized, it could leak private information.

This is basically what happened with Heartbleed [60], a bug that was unknowingly introduced in 2012 in the OpenSSL program, and disclosed and patched in 2014. In short, the client would send a word and its length to the server, and the server would store the word in memory then send back the length of the word from said memory. However, the server never checked that the length was actually that of the word. As such, it was possible to request 500 bytes when having sent 4, and to get back the word, and the next 496 bytes from the server's memory. This was notably used to steal Social Insurance Numbers in Canada.

When a bug can be actively exploited by an attacker to change the behavior of the program, or retrieve information, we call it a *vulnerability*.

As for *crashes*, they are merely a possible symptom of a bug. For example, with the out-of-bounds example above, if the code had crashed, that would have alerted us to the presence of a bug.

Proving vs Testing It is possible to *prove* that a program behaves as it is supposed to, and hence does not contain bugs [18]. This requires for the developers to first define a specification: "what the program is supposed to do", then verify certain properties wrt the specification and the code. For example, in Software Model Checking [17, 53], one would define a model of the program's

behavior, then analyze said model to ensure some properties are satisfied, such as mutual exclusion - two threads of execution cannot simultaneously access a critical section. It is also possible to achieve such proofs on the final code. For example, Frama-C [42, 3] has a WP [12] plugin, which will prove that any execution of the code will satisfy a user-defined contract, using weakest-precondition calculus [25].

As efficient as those techniques are, they are expensive. They require the developers to be familiar with both the technique and the program, in order to design an accurate specification and then prove that the program is correct wrt the specification, which also takes time.

Alternatively, it is possible to *test* the program [1]. When testing, the program will be run on a set of test cases, called a test suite, and the results of the execution will be analyzed. Again, there are different ways to test a program. Unit tests will check whether a particular fragment of the code behaves as expected, by running it on a given set of inputs, and comparing the results to the expected ones. For more global tests, system testing will check whether a complete system satisfies its specification.

In this thesis, we are working with automated testing. We want to automatically generate a test suite, with the goal being not to compare the behavior to a specification, but rather to detect any crashes, thus bugs. To achieve this, we need the test suite to cover as many of the program's execution paths as possible. Nowadays, this is usually done using one of two techniques: symbolic execution [41, 11, 36] or fuzzing [52, 49].

Symbolic Execution and Fuzzing *Symbolic Execution* (SE) [41, 11, 36] aims to explore a program by analyzing every execution path, and crafting a test case for each one. It is the combination of two components:

- the *symbolic execution engine* will analyze the paths, following one of several strategies (Depth-First Search, Breadth-First Search, random, etc). For each path, it will infer a path predicate by semantically analyzing the instructions in the path. A path predicate is a constraint on the input such that any test case satisfying the predicate will follow the path.
- the *SMT solver* [44] will take a path predicate and create a solution, if one exists.

Symbolic execution can be used to create an extensive test suite by crafting a test case for each path, at the cost of analyzing every path and solving the resulting path predicate.

Fuzzing [52, 49], on the other hand, aims to run the program on many test cases, in order to hopefully trigger all possible paths. As such, it relies on

quick and easy generation of test cases. The most basic fuzzers, referred to as blackbox fuzzers, simply generate random test cases, without having any knowledge about the Program Under Test (PUT). As it became clear that this was not enough to fully test programs, for example ones that expect a specific input format, smarter fuzzers were designed. Called greybox fuzzers, they often add an analysis in order to get some information about the PUT, without slowing down the process. For example, AFL [69] is able to detect whether a new test case reached an unexplored part of the program. Such test cases are deemed interesting, and used as the basis to create future ones.

Problem, goal, challenges

By analyzing complete paths, however long they are, symbolic execution is able to explore arbitrarily deep parts of the program. On the other hand, in order to fully explore the program, it will have to systematically analyze each path. And the bigger the program is, the more paths it has: this is a phenomenon called *path explosion*. As a result, symbolic execution does not scale well on large programs. It is also dependent on SMT solving ability to solve the constraints. To avoid these pitfalls, users might have to bound the symbolic execution, limiting the number of paths explored, or drop difficult constraints.

When it comes to fuzzing, its cheap efficient generation of test cases is also its weakness. In particular, test cases are generated without the fuzzer taking into consideration the semantics of the program. Consequently, the fuzzer's efficiency does not depend on the program's size or complexity. But this also leaves the fuzzer blind to potentially interesting information about the internals of the PUT. For example, if the program contained a condition such as `buf[0] == 0xdeadbeef`, the test generation engine would just create test cases until it randomly set `buf[0]` to `0xdeadbeef`, something highly unlikely.

The weaknesses and strengths of fuzzing and symbolic execution appear to be rather complementary: we would want a tool able to pass hard conditions while quickly explorating easier paths.

Our objective is precisely to develop an automated test generation technique, combining the reasoning power of symbolic execution with the light cost of greybox fuzzing.

More precisely, we want to build an *efficient* approach, both able to reason about complex code and to generate test cases quickly. This approach would combine symbolic reasoning with a fuzzer, and we identify five challenges. The symbolic reasoning would need to be (1) cheap – no SMT solver, (2) targeted

to interesting paths, (3) correct, while the fuzzer would need to be (4) efficient. Finally, we want both techniques to be deeply integrated with one another (5).

Several recent works [59, 68, 39, 14, 15] follow roughly the same goal, but none of them satisfy all the objectives listed above. Many of these approaches [59, 68] combine an off-the-shelf fuzzer together with an off-the-shelf symbolic executor, i.e. they do not integrate the two techniques at the *conceptual* level. In addition, the analyses performed by many of these tools [39, 14, 15], as well as their properties, are often loosely defined, leading sometimes to public criticism about their actual soundness[1]. In this work, we aim at introducing a *correct* test case generation technique, which genuinely *integrates* the concepts from symbolic execution with those of fuzzing.

Our Approach

The solution we propose to this problem combines two novel ideas: *Lightweight Symbolic Execution* and *Constrained Fuzzing*.

- *Lightweight Symbolic Execution* (LSE) is a symbolic analysis. Similarly to symbolic execution, it will analyze a trace and return a path predicate. However the path predicates are different, because they are expressed using a fragment of symbolic execution's usual target language. By thus restricting the complexity of the solution space, the path predicates generated by LSE are *easily-enumerable*. While this makes the inference of the constraints more complicated, it means enumerating solutions can be done without resorting to an SMT solver;
- We combine this analysis to a *Constrained Fuzzer*. While for symbolic execution an SMT solver would create a test case targeting a full path, our constrained fuzzer will create multiple test cases targeting a branch in the path. This allows us to first guide the fuzzing past any difficult condition, and then cheaply explore multiple paths starting at this condition. We also use fuzzing's feedback to guide the LSE towards interesting paths, rather than having to analyze every path.

Overall, LSE's reasoning power will lead the exploration past specific conditions and towards interesting parts of the code, while the constrained fuzzer will efficiently create test cases, including solutions to the constraints. This allows us to explore the program without systematically relying on symbolic analysis, and removes the need for an SMT solver to create test cases satisfying the constraints.

[1] https://andreas-zeller.blogspot.com/2019/10/when-results-are-all-that-matters-case.html

Contributions and outline

Contributions As a summary, our contribution is three-fold:

- We propose *Lightweight Symbolic Execution* (Chapter 4), a novel variant of symbolic execution. Using abstractions, it creates path predicates that are *easily-enumerable* (Section 4.2). To achieve this, it sacrifices completeness in exchange for fast enumeration of solutions, removing the need for an SMT solver. We also describe how we correctly infer such constraints (Section 4.4), as well as how to enumerate solutions (Section 4.2). We implemented this method as part of the BINSEC tool [26, 24];

- We present *Constrained Fuzzing* (Chapter 5), a modification of Greybox Fuzzing tailored for combination with lightweight symbolic execution, which creates solutions to easily-enumerable path predicates (Section 5.2). Both tools communicate through traces (Section 5.3) – to guide the LSE's analysis – and easily-enumerable path predicates (Section 5.4)– to guide the fuzzing's test generation. We modified AFL [69] and combined it to BINSEC in order to create CONFUZZ;

- We evaluated the resulting tool on the LAVA-M [28] benchmark (Section 5.5), a set of binaries extracted from GNU Coreutils in which vulnerabilities were artificially injected. We compare ourselves to the state of the art in terms of fuzzing and symbolic execution, as well as some tools that combine both, with regard to coverage and bug finding abilities.

Outline The rest of this document is organized as follows:

Chapter 2 introduces program analysis, and how it can be used to automatically generate test suites

Chapter 3 presents as a motivating example a sample program which causes path explosion in symbolic execution and contains conditions fuzzing struggles solving

Chapter 4 describes Lightweight Symbolic Execution, and defines the underlying principle of easily-enumerable path predicates. It also shows how we infer such constraints from a given trace

Chapter 5 describes Constrained Fuzzing, and how it is combined with Lightweight Symbolic Execution to generate solutions to easily-enumerable path predicates, as well as guide the symbolic analysis

Chapter 6 presents our experimental evaluation on a standard fuzzing benchmark, comparing our tool to the state of the art

Chapter 2

Background

Contents

In this chapter we introduce several concepts that are used in this thesis.

We first present an overview of program analysis, what it can achieve and some of the techniques used to analyze programs (Section 2.1), in particular automated test generation. We then introduce two of such techniques which are key to this work: symbolic execution (Section 2.2) and fuzzing (Section 2.3).

Finally, we discuss the challenges of binary code analysis (Section 2.4).

Overview of Program Analysis

While traditional programs could take a number, or a file, as input, program analyzers reason about other programs.

Programs have always contained bugs, but as computer programs became widespread, techniques were designed to either find said bugs, or avoid them altogether. We call such techniques program analysis: instead of taking a number or file as input, they analyze other programs. Formal Methods [18] in particular were developed in the 70s. By using mathematics to reason about programs, they aim to prove properties such as the absence of runtime errors. Furthermore, program analysis can also be used to optimize a program during compilation.

In this section, we will present different reasons and ways to analyze a program.

What Do We Analyze?

When someone says "analyze a program", the word program can have different meanings. In particular, different representations of a program can exist, and each can be analyzed. The following are some of the options.

Model Before writing a single line of code, it is possible to define a formal specification of a program. This can be used as a basis to build a model representing the program, which can then be analyzed [17, 53] or used to generate tests [22, 47]. Whether the conclusions of this analysis still apply to the finished program depends on how accurate the model was, and if any changes were made while writing the code;

Source Code Once the development of a program has started, source code is produced. It is possible to directly analyze this source code, and draw conclusions on the expected program's behavior. However, said behavior may change depending on the compiler [2]. For example C code might contain Undefined Behaviors – pieces of code which were not defined in the C standard – for which there are no compiling rules;

Binary Code A finished program has to be compiled in order to be run on a computer. The result is an executable file, made of 0s and 1s. This can be analyzed by specialized analyzers, capable of disassembling the binary code into a manageable intermediary representation. Disassembling is not a trivial step, and it is nearly impossible to retrieve every possible piece of information – for example types – from the source code [51, 57, 45].

How Do We Analyze?

We can differentiate between two types of analyses: static [29] and dynamic [38]. *Static analyses* reason on the program's code, or model, without executing it. On the other hand, *dynamic analyses* rely on running the program on test cases, and reason on the resulting execution traces. For example, when looking at the program in Figure 1.1, we could consider the input as an abstract undetermined value, and try to look at all the possible paths, forking the analysis when there are conditions. We could also execute it with multiple concrete inputs, leaving the analyzer with a single possible path for each run. This removes any forking problems, since the result of a condition is always known. It also allows the analysis to access the concrete value of some operations, such as external library calls.

Why Do We Analyze?

Analyzers can serve multiple purposes, among which are *proving* properties about the program, or *testing* it.

Proving means proving properties about the program. Possible properties can be mutual exclusion, or the absence of run-time errors. It requires first to formally define the program's specification – what it is meant to do. Different techniques offer different results, such as:

Model Checking [53, 17] is a technique that can be used as early as the design phase, before the program is implemented. The specification is directly used to model the expected program's behavior, which is then analyzed to verify properties. For example, it is possible to model a distributed system, with different components that interact together, and formally verify that two components cannot simultaneously access a critical section of memory, aka mutual exclusion;

Weakest Precondition Calculus [25] is a technique used for example by the C analysis framework Frama-C [3, 12]. It works by defining a pre-condition and a post-condition for each part of the code, and it will verify that given the pre-condition, the post-condition is valid wrt the targeted code. For example, we could add conditions to the example from Figure 1.1, verifying that we never access memory out of the bounds of the table. In this case, the verification fails for the second loop, since it does go beyond the end of the table. This would have alerted us to the presence of a bug in the program.

Abstract Interpretation [20] is a technique used with various goals, such as checking the absence of run-time errors [21]. The key idea is to represent memory states in an abstract way, which we can reason about using specific operations that allow us to propagate said representation. This information can then be used to infer information about the program, depending on the goal of the analysis. For example, we can abstract memory states as intervals of values, an abstraction which is updated when instructions modify the value of the memory state. In this case, if you divide by x, and abstract interpretation determined that $x \in [4; 10]$, you are certain that there will not be a division by zero. Since abstract interpretation reasons about an over-approximation of the memory states' values, any property that is true for the program is true for its abstraction, though there might be false positives: properties that are true for the abstraction and not the program.

No matter the technique used, proving properties on a program requires having people familiar with both the program and the intended proof technique. This means either out-sourcing to specialist consultants, hiring someone that already has this knowledge, or training developers so they can take on both roles. In any case, it is expensive for whoever is developing the program, but it offers formal guarantees about the behavior of the program.

Testing [1] does not offer such formal guarantees. The main idea of testing is to run the program on test cases, and compare the outcome to what was expected, be it "the program does not crash" or "when given said input, the program returns said result". There are various ways to test a program:

Unit Testing is when someone, possibly the developer, creates a set of tests for small parts of the programs. This is used to check whether individual functions return the expected result. For example, if we had a function that returns the Fibonacci value for a number, we might want to test that it returns an error when called with a negative number, 0 for 0, 1 for 1, 1 for 2, and maybe one or two higher numbers. Since test cases are hand-crafted and associated with the expected results, this is done by a person, which will usually check corner cases and a few regular cases.

Automated Testing [49, 36, 11] aims to check whether there are any inputs which trigger a crash. To achieve this, the program will be analyzed by a tool, which will automatically generate a test suite. Usually the goal will be to maximize the coverage attained by the test suite, in order to make the exploration as exhaustive as possible. While it is impossible to create

a tool that will always achieve full coverage for any program – mostly because it is sometimes impossible to achieve full coverage, for example if the program contains infinite loops – it takes the pressure off of the developers. Instead of having someone hand crafting a test suite, they will just have to run the chosen generation technique on the program, and then run the program on the resulting test suite. As such, while automated testing does not guarantee that the program does not contain bugs, it is still efficient at finding bugs, while remaining cheap. This makes it a good way to supplement other testing techniques, when one does not want to use formal proof.

In both cases, the end goal of the analyses varies whether we are considering the safety or the security of the program. *Safety* answers the question "does the program work as it should?" while *security* answers the question "could an attacker take advantage of the program?". For example, if the program crashes, safety is not satisfied: the program is not supposed to do that. The crash could be harmless, or it could be abused by a malicious user, as in Figure 1.1. Only in the second case is it considered a breach of security. Nevertheless, finding safety faults is a good way to check for openings that attackers could exploit, and is thus a step to ensure security.

ConFuzz In our case, we analyze binary code, which has been instrumented at compile-time, allowing us to work independently from the source code. Our goal is to automatically test the program in order to find bugs, and we achieve this by combining fuzzing – in order to generate test cases – and dynamic symbolic execution – in order to explore new parts of the code. Furthermore, as we search for crashing test cases, we concern ourselves with the safety of the program, letting further analyses determine whether the bug might be a vulnerability.

Symbolic Execution

Symbolic Execution (SE) [41, 11, 9, 58] acts similarly to an interpreter, in that it will go over each instruction of the program and apply its semantics on the given input. The difference is that, where an interpreter would consider concrete values as input, symbolic execution will consider symbolic values. Along the execution, the SE engine will maintain two pieces of information about the state of the program: a *symbolic state* Σ – a map binding variables to their symbolic value – and a *path predicate* φ – a predicate over the symbolic variables, describing the condition for a test case to reach the current instruction. When

an input is provided by the user, the engine represents the data with a symbolic variable, meaning "this could be anything". On branching instructions, since there is no unique possibility, symbolic execution forks in order to explore all possible paths. It chooses which branch to explore first depending on a user-defined strategy (Depth-First Search, random, etc.)

When the analysis reaches the end of a path, the resulting path predicate is a constraint over the input. If we were to run the program on any solution of the predicate, the concrete execution would follow all the branching choices made by the symbolic analysis. Such a solution will usually be generated by sending the predicate to an off-the-shelf solver. If the constraint has a solution, the solver will return a test case which covers the path. If there is no solution, it means that the path is unfeasible.

The execution tree on the right of Figure 2.1 shows the symbolic state and path predicate for each of the (numbered) instructions in a sample program. In this tree, x_0 is the symbol corresponding to the user input returned by the read_int function, and forking happens due to the condition **if** (x >=5).

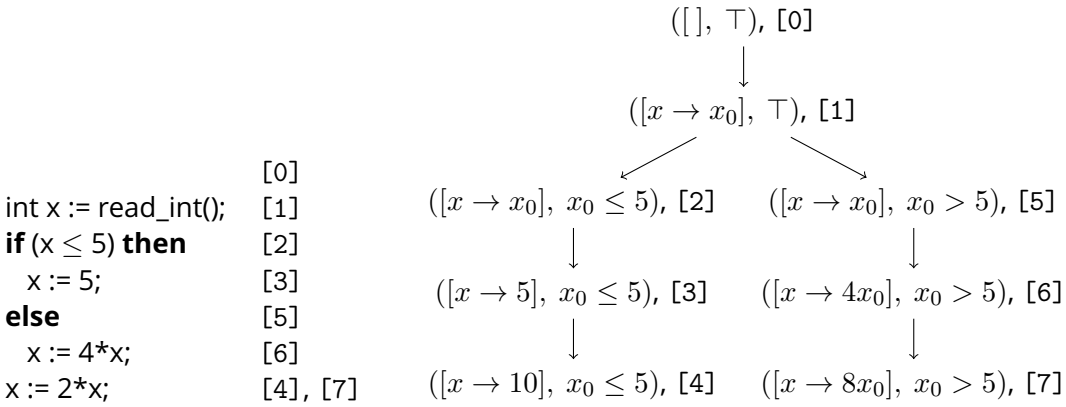


Figure 2.1: Symbolic execution of a sample program

Formalization Given a program under test P over a set of input variables X, a seed t is a valuation of every variable in X. The execution of P over t, noted $P(t)$, follows a path $\sigma \triangleq \{l_0, ..., l_n\}$ where the different l_i are instructions of P.

Definition 2.2.1 (Perfect Path Predicate)**.** *A predicate φ_σ over X is a* perfect path

predicate *for path σ if, for any seed t, we have:*

$$t \models \varphi_\sigma \Leftrightarrow P(t) \textit{ follows } \sigma$$

It is not always possible to compute perfect path predicates. A path predicate is *correct* if all the predicate solutions are seeds that do cover the intended path (left-to-right implication, under-approximation), and *complete* if any seed covering the path is a solution (right-to-left implication, over-approximation).

Figure 2.2 shows an example of this. Let us imagine φ_1 is a perfect path predicate for the target path. Then φ_2 is correct, but not complete: any solution of φ_2 is a solution of φ_1, but for example $\{x \mapsto 9\}$ is not included. By over-constraining the solution space, we lose possible solutions. On the other hand, φ_3 is complete but not correct: any solution of φ_1 is a solution of φ_3, but $\{x \mapsto 6\}$ is a solution of φ_3 that is incorrect for φ_1. There are solutions of the path predicate that do not actually follow the path.

$\varphi_1 = 2 \leq x < 5 \vee 8 < x \leq 10$
$\varphi_2 = 2 \leq x < 5$
$\varphi_3 = 2 \leq x \leq 10$

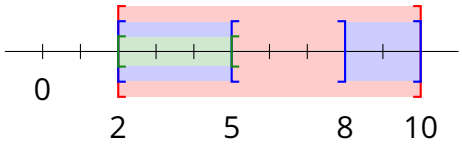


Figure 2.2: Example of correct and complete path predicates

Dynamic Symbolic Execution

If Symbolic Execution is unable to add a condition to the path predicate, the symbolic engine will have to drop the current path. This can happen for several reasons, such as the condition relying on code that is not accessible – e.g. a call to an external library – or the condition being too complicated for the solver – e.g. Mixed Boolean Arithmetic. Such interruptions of the analysis will prevent it from reaching full coverage, as it cannot explore every path of the program.

In order to prevent this, several techniques propose to mix symbolic and concrete executions. Such tools are referred to as "Dynamic Symbolic Execution" [11].

Execution-Generated Testing [9, 8] When executing the program, the engine will differentiate between expressions that have a purely concrete value, and symbolic expressions. As such, operations that only operate on concrete values will be dynamically evaluated, as they are in the original program. If there

is at least one symbolic operand, the operation will be evaluated symbolically, in the same way as Static Symbolic Execution. This means that calls to external libraries, when their arguments are concrete, are directly executed rather than considered symbolic. Similarly, if conditions, however complicated, are concrete, the outcome is computed without calling the SMT solver. This helps decrease the number of times symbolic execution will have to abandon a path because it cannot reason about it, without completely solving the problem.

Concolic Testing [58, 35] Instead of reasoning on the whole program, Concolic Testing reasons on a single execution path at a time. By running the program with concrete input, it gets the trace – the list of executed instructions. The analyzer will then follow the path, keeping the symbolic state of the variables as well as their concrete state. The concrete state can then be used as needed, for example for library functions results. In order to explore more paths, every time the analysis encounters a condition with an unexplored branch, it will try negating it and adding it to the path predicate in order to create a test case that takes this new branch. For example with the program in Figure 2.1, it could start with $\{x \mapsto 0\}$, which will take the $x_0 \leq 5$ branch. It will then add the negation, $x_0 > 5$, to the (empty) path predicate, and solve it to create $\{x \mapsto 6\}$ and explore the second branch. As there are no more unexplored branches, the engine would conclude the analysis to be complete, and stop there. It solves the problem of unavailable code and complicated conditions by directly giving them their concrete value in the current execution, at the cost of restraining those expressions to a single value.

KLEE, State of the Art of Symbolic Execution

KLEE [8] is the state of the art of symbolic execution. Its goal is to assist in testing a program by automatically generating test cases that reach high-coverage. They compared themselves to the line coverage achieved by the developer's hand-written tests, and found that their tests were more efficient, thus proving the efficiency of automatic test generation. They achieve this by leveraging symbolic execution and making it more robust and scalable, using constraint solving optimization and search heuristics.

Difficulties

Path Explosion As mentioned in the introduction, path explosion happens when the number of paths in a program grows to the point that it is impossible to explore every path in a reasonable time. To prevent the analysis from running forever, users will usually bound the symbolic execution, be it by giving it a time

budget, a limited number of paths to explore, a maximum depth to explore, etc. As a result, the engine will not be able to explore everything, and should explore the most relevant paths first. This can be done by using heuristics to prioritize certain paths [7] usually based on criteria such as the instruction and branch coverage. It is also possible to use program analysis to soundly decrease the complexity, for example by merging paths that have the same outcome [43], or pruning redundant paths [6, 50].

Constraint Solving Symbolic Execution heavily relies on SMT solvers. Satisfiability Modulo Theory is the problem of whether a formula in first-order logic admits a solution. SMT solvers were built to answer this question, by combining SAT and other theories. In our case they are used to solved path predicates, but they can also be used in WP calculus [12, 31], in model-checking [19], etc. However SMT solvers are not perfect and there are constraints they cannot solve. The goal with Symbolic Execution is to prevent this from happening, as much as possible. As such, the analysis will often try to simplify the constraint solving. Two such optimization are:

irrelevant constraint elimination aims to simplify the constraint by removing terms that do not influence the result. For example, when encountering a new condition, SE will check if either of the path is feasible by adding the condition and its negation – separately – to the current path predicate. At this point, any constraints already part of the predicate that are independent from the new condition will not influence the result, and can thus be safely removed in order to make the resolution easier.

incremental solving aims to reuse past information as much as possible. This is useful because usually many paths share similar constraints. In the case of KLEE, this is done by keeping a map between each previously solved constraint and a single solution, if there is one. This solution can be used not only when re-analyzing the same constraint, but also a subset or a superset. If we are trying to solve a subset, then the original solution is still a solution, while when trying to solve a superset, we can quickly check whether the solution is still valid for the new constraint.

Memory Models [40, 30] One of the key component of symbolic execution is its memory model. A memory model is the way the developers of the execution engine have decided to represent the memory and the data from the program. It is used to translate program instructions into symbolic constraints, and as such heavily influences the coverage the program can attain, as well as its scalability. A trade-off needs to be made between the analysis' efficiency

and precision. For example, in a program, integers would usually be fixed-width integers. While it is possible to represent them using mathematical integers, which would make the analysis easier, this hides some corner cases, such as arithmetic overflow. A symbolic analysis that uses this representation will not be able to explore paths made possible by an arithmetic overflow, and might not find some bugs. As a result, it is advised to tune the memory model to the program to be analyzed, which again requires to have someone familiar with both the PUT and the analysis.

Concretization [35, 23, 34] While the concretization techniques used in Dynamic Symbolic Execution help mitigate some of the issues Static Symbolic Execution faces, they do so by sacrificing the completeness of the path predicate. Indeed, by restraining a function output to its result in a concrete execution, we transform a possibly complex function into a blackbox with only a single output. As a result, we might make impossible paths that were possible in the original program, and which we could have explored with Static Symbolic Execution.

Fuzzing

Fuzzing [52, 49] is a simple yet very effective automated testing technique. Its goal is to explore the program's execution paths by running it on many test cases.

Information About Program As usual when it comes to testing, we differentiate between blackbox, whitebox and greybox analyses. Blackbox means the analysis has no knowledge about the PUT except for the execution results, whitebox means it has full access to the code, and greybox is anything in between. When it comes to fuzzing, *blackbox fuzzers* [65, 64, 66] are the first and most basic of fuzzers: they simply generate random test cases and run the program, observing the outcome to determine whether it crashed. They are fast, but blind. In contrast, *whitebox fuzzers* [36] mainly employ heavy-weight program analysis like symbolic execution to explore all the feasible paths of a program. *Greybox fuzzing* [69] lies in-between: it uses only light-weight program analysis or feedback information to guide the search.

Test Case Generation Another key to fuzzing is "how are the test cases generated?". We need the test generation to be efficient, but ideally we would want the test cases to follow the PUT's expected format, so as to explore past the program's initial format checks. Most fuzzers, at least those that do not just generate random test cases, apply one of two techniques:

Grammar-based fuzzers [64, 66] will initially receive a grammar or input model that precisely describes the expected format. This ensure that any test case will be valid, and is especially useful when fuzzing precise programs, such as an interpreter [62]. To get the proper grammar for each fuzzing target, it is however necessary to analyze it, prior to fuzzing. And while there are some techniques aiming to automatize this with machine learning [37], it remains time-consuming.

Mutation-based fuzzers create new test cases by mutating existing ones. The idea is that if you have a valid test case and do not modify it too much, the result will probably be valid as well. Ideally, such fuzzers would initially be given seed test cases that are valid, as a basis to mutation. However it has been proven that some fuzzers are able to create a valid test case from scratch, and then mutate it to explore the code further. While it is efficient, as proven by AFL, we would expect similar test cases to explore the same part of the code, making exhaustive exploration more complicated.

Goal Finally, fuzzers act differently depending on their goal. Again, we consider two types of fuzzers: directed and coverage-based. *Directed fuzzers* [4, 13] target a specific pattern, or part of the code, or type of bug. As such, they specialize in their target, and do not consider anything else. While these fuzzers are very efficient when one has a specific purpose in mind, such as checking patches [67] or looking for Use-After-Free bugs [54], they are not suited for general bug-finding. On the other hand, *coverage-based fuzzers* aim to maximize the code coverage achieved by the test suite. To achieve this, they might focus the exploration on paths that lead to new parts of the code, or deeper ones. This makes them the go-to fuzzers when doing automated testing.

AFL, State of The Art Greybox Fuzzing

In this section, we will present AFL [69], a fuzzing tool developed by Michal Zalewski. Its trophy case [63], as well as the number of tools [46, 59, 55, 48, 33] based on it, including this one, clearly place it at the top of the state of the art of greybox fuzzers.

AFL is a *coverage-based mutational greybox fuzzer*. In particular, it uses coverage information to guide its test case generation. As illustrated in Figure 2.3, AFL keeps a test case queue which initially contains the seed test cases. It then creates new test cases, as follows:

- one test case will be selected, then mutated, in order to create a new test case ;

- the PUT, instrumented either at compile-time or on-the-fly, is executed on the test case, and the output is observed to determine whether the test case triggers a crash ;
- through the instrumentation, AFL retrieves the branch coverage of the new test case, and compares it to the branches covered so far to determine if the new test case is *interesting* – understand, leads to a new part of the code. If it does, it is added to the queue ;
- a new test case is selected, until the user terminates AFL.

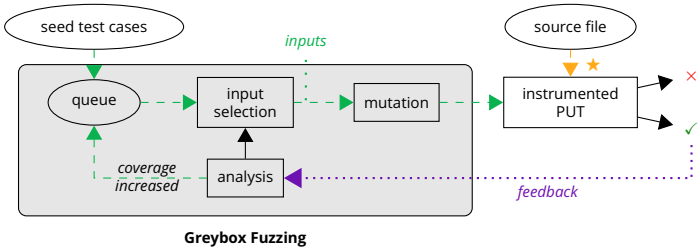


Figure 2.3: Representation of a coverage-based greybox fuzzer. ★: instrumentation

Difficulties of fuzzing We can distinguish two kinds of difficulties encountered by fuzzers:

Complex structures The randomness behind fuzzing means that it has a low probability of finding a solution to hard code such as magic byte comparisons or parsing, which usually depend on the input.

Code coverage In direct relation to the previous problem, fuzzing sometimes explores only the surface of the program and cannot explore deep paths in the PUT.

Binary Analysis

Context

For a program to be understood by a computer, it must first be compiled into a specific file format. In most cases, this is binary code: a series of 0 and 1s, that the processor can interpret. As a human, we can use a disassembler to translate those bits into assembly code, a very basic language that is a direct representation of the machine's operations.

In the same way that there are different programming languages (C, OCaml, Python, Java, etc.), there are different binary file formats, each with its own assembly language. This does not depend on the program or the developer, but on the platform the program is run on. For example, Intel uses x84 (resp x64) for its 32 (resp 64) bits processors.

Figure 2.4 shows an example on an extract of Figure 1.1. The C code was compiled into x86 binary code using `gcc -m32`, then disassembled using `objdump -d`. We added comments to the disassembled code to show how it translates back into pseudo C code. We notice that the code has been broken up into simpler operations, and in particular the loop is executed through jumps. The cmp operation will compute `4 - cnt` then update flags in memories depending on whether the result was zero, or if there was a carry, or an overflow, etc. The next instructions, `jle` will jump depending on the content of the flag, here if `4 <= cnt`.

```
for (cnt = 0; cnt < 5; cnt++)
  tab[cnt] = cnt+1;
```

C code

```
00000480: 0489 45d4 65a1 1400 0000 8945 f431 c0c7
00000490: 45dc 0000 0000 eb11 8b45 dc8d 5001 8b45
000004a0: dc89 5485 e083 45dc 0183 7ddc 047e e9c7
```

Binary code

```
c7 45 dc 00 00 00 00   movl   $0x0,-0x24(%ebp)          cnt = 0
eb 11                  jmp    80484a9 <condition>       goto condition
                                                        loop:
8b 45 dc               mov    -0x24(%ebp),%eax          eax = cnt
8d 50 01               lea    0x1(%eax),%edx            edx = eax + 1 //cnt + 1
8b 45 dc               mov    -0x24(%ebp),%eax          eax = cnt
89 54 85 e0            mov    %edx,-0x20(%ebp,%eax,4)   tab[cnt] = edx //cnt + 1
83 45 dc 01            addl   $0x1,-0x24(%ebp)          cnt = cnt + 1
                                                        condition:
83 7d dc 04            cmpl   $0x4,-0x24(%ebp)          cmp 4 cnt
7e e9                  jle    8048498 <loop>            if (4 <= cnt) goto loop
```

Disassembled code – using objdump – and comments

Figure 2.4: Example of compiled / disassembled code

When analyzing a finished program, there are two options. It is possible to analyze the source code of the program, as Frama-C does with C code, or the binary code. Binary analyzers are often a combination of disassemblers and analyses. The disassembler will translate a program from a given format into an intermediary representation, and then analyses will be applied to it, depending on the goal.

There are mainly two reasons to analyze a binary rather than the source code:

- the person analyzing the program does not need access to the source code. This comes into play when analyzing programs whose source code

is not available, be it confidential software, off-the-shelf components, malware, etc.

- the analyzed code is what gets actually executed. For example, when C programs contain Undefined Behaviors, these might get compiled differently depending on the compiler and its optimization level. In the case of the program in Figure 1.1, we get `1-2-3-4-5-32764` on the lowest optimization, but higher ones set the out of bounds access to 0. If instead of printing, we were dividing by `tab[cnt]`, this would result in a crash.

Challenges

As other research works have explored before, correctly disassembling and analyzing binary code is not a trivial task [57, 51].

Disassembly The first challenge when analyzing binary code is to correctly interpret the sequence of 0 and 1s. This requires starting at the correct entry-point, then correctly matching bytes sequences to an operation, depending on the file format. There are several ways to achieve this:

linear disassembly disassembles every byte in the program, one set at a time, in the order they are written in

recursive disassembly follows along the execution paths, mainly by resolving jump targets, and disassembles the executed code – with the caveat of dynamic jumps that cannot be resolved

While recursive disassembly might seem more accurate, it depends on whether it is possible to solve jump targets. On the other hand, programs are usually made in a way that makes linear disassembly easy, for example by aligning instructions.

Loss of information Compared to source code, assembly code is much simpler. This is because the compiler does not deal with conditions, or long operations, or types. In particular, data is merely a set of bytes, stored in registers of fixed size, and conditions become a conditional jump depending on the flags. For example, in Figure 2.4 we see that the loop is translated by a test on `4` and `cnt`, followed by a conditional jump out of the loop. More often than not, the only information the analyzer has access to is the one present in the binary code. This obviously makes the analysis more complicated: the signedness is essential when trying to interpret the values, while the size indicates which part of the data is actually relevant, as opposed to filler bits. The analyze needs to

deduce this type information from the operations used [45], something that is not always possible. And while higher-level conditions can be recovered [27], it does require additional work.

Chapter 3

Motivating Example

Contents

In this chapter we present the difficulties of fuzzing and symbolic execution on a small example, as well as ConFuzz's solution.

First we introduce the code of our example, which contains both a loop – thus a high number of possible paths – and nested conditions – thus hard-to-solve constraints (Section 3.1). Then we explain how AFL [69] and KLEE [8] react to both parts of the code (Section 3.2), before explaining our own approach (Section 3.3). Finally, we show the results of all three tools, with varying number of loop iterations and nested conditions (Section 3.4).

Code

We illustrate the difficulties of symbolic execution (resp. fuzzing) by looking at the performance of KLEE [8] (resp. AFL [69]) on the sample program from Figure 3.1.

This program contains a loop – with 0, 10 or 20 iterations – and nested conditions – with 0, 3 or 5 conditions.

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#define BUF_LENGTH 64

int main(int argc, char** argv) {

  char buf[BUF_LENGTH];
  int x, y;

  int res = read(0, buf, BUF_LENGTH);

  if (res < BUF_LENGTH) {
    printf("entry too small\n");
    return 0;
  }

  /* Loop */
  int cpt;
  for (cpt = 16; cpt < 36; cpt++) {
    if (buf[cpt] == cpt % 20)
      y += 1;
  }
  printf("%i\n", y);

  /* Nested conditions */
  if (buf[0] == 'a')
    if (buf[4] == 'F')
      if (buf[7] == '6')
        if (buf[12] == 'g')
          if (buf[15] == 'L')
            x = 1;
          else
            x = 2;
        else
          x = 3;
      else
        x = 4;
    else
      x = 5;
  else
    x = 6;
  printf("%i\n", x);

  return 0;
}
```


Figure 3.1: Sample program

Using State-of-the-Art Tools

Fuzzing with AFL

In order to fully explore the program, we need to generate test cases for both key parts: the loop, and the nested conditions. In this section, we explain how fuzzing deals with both type of difficulties, and especially how conditions pose an obstacle.

Loop AFL does not have any knowledge about the Control Flow Graph (CFG) of the PUT. This means that when it comes to path exploration, fuzzing does not target paths, but merely relies on the coverage information to identify interesting test cases. Namely, it does not try to explore every paths of the loop, but simply generates test cases that happen to satisfy the conditions of the loop.

In this case, since the conditions are not nested, they can be satisfied independently from each other. The probability of finding the solution for each one is 2^{-8}, something that has a low impact on AFL's performance. In practice, we observe that there are some variations depending on the number of iterations, but they are insignificant compared to the variations depending on the number of conditions. Most likely than not, those only happen because AFL is non-deterministic, so the output varies for each run.

Nested conditions For nested conditions, AFL needs to create a test case that satisfies all of the conditions. Let us imagine we start our fuzzer with an initial test case t_0 made of 0s, and ignore the loop. AFL might do the following:

1. mutate the initial test case until it creates $t_1 \triangleq$ `"a0..."`. Since it is interesting, add it to the queue.
2. alternatively mutate t_0 and t_1 until it creates $t_2 \triangleq$ `"a000F0..."`. Since mutations are blind to what makes a test case interesting, it might mutate the first byte of t_1, and create multiple test cases that now break the first condition.
3. keep mutating t_0, t_1 and t_2 until it creates $t_3 \triangleq$ `"a000F0060..."`
4. and so on until it creates a test case for each condition

If we did not have coverage information, we would not be able to identify interesting test cases, and would have to create the final test case by directly mutating t_0. Nonetheless, AFL still struggles finding the solutions to the conditions. We observe that AFL takes longer to explore every paths when the number of nested conditions increase.

Symbolic Execution with KLEE

Fuzzing struggles with finding solutions to conditions, even more so if the conditions are nested. On the other hand, KLEE tries to generate a test case for every path. In particular, here, the most iterations the loop has, the longer KLEE spends trying to explore everything.

Loop When it comes to the loop, KLEE actively tries to explore every possible paths. As such, it will try to find a path where the condition is verified only for 0, one where it is verified for 0 and 1, 0 and 2, 0, 1 and 2, etc. This leads to what we call path explosion: the number of paths considered by KLEE grows exponentially. Given the time taken by KLEE to create the path predicate then solve it using an SMT solver, this path explosion has direct consequences on KLEE's performance, as we can observe when modifying the number of loop iterations.

Nested conditions Nested conditions pose no problem to KLEE. The symbolic execution will create 6 path predicates, one for each path, and they will instantly be solved by the SMT solver. In practice, we observe that the performance does not depend much on the number of nested conditions, though we do observe a significant difference when a high number of conditions is combined to a high number of loop iterations. This is because adding conditions add paths for each of the possible path through the loop.

Our approach

Using a combination of Lightweight Symbolic Execution and Constrained Fuzzing allows us to solve both these issues.

Since we rely on the Constrained Fuzzer to create solutions and guide the symbolic execution, we deal with the loop in a way similar to AFL. Rather than aiming to explore every path, we rely on the coverage information to guide us towards the interesting paths. This allows us to quickly explore the loop, no matter the number of iterations.

As for the nested conditions, the Lightweight Symbolic Execution will create easily-enumerable path predicates for each path. This will both lead the Constrained Fuzzer past those difficult conditions, and guide further fuzzing towards this new part of the code. Indeed, instead of blindly mutating $t_1 \triangleq$ `"a0..."`, it will create solutions to the predicate $\widetilde{\varphi}_1 \triangleq t[0] = \texttt{'a'}$ on t_1, hence always satisfying the first condition.

In practice, ConFuzz is only mildly affected by the number of iterations or conditions. As a result, it outperforms AFL every time, has similar results to KLEE for 10 iterations, and is 150 times better when there are 20 iterations.

Results

We present in this section the results from running AFL, KLEE and ConFuzz on the motivating example. To illustrate each tool's strength and weakness, we vary the number of iterations (0, 10 or 20) and nested conditions (0, 3 or 5). This gives us 9 different results for each tool.

Furthermore, to account for the non-determinism of the tools – mostly AFL and ConFuzz, we run each tool 10 times, with a long enough timeout to explore everything. The average results for each configuration are presented in tables 3.1 to 3.3. For each configuration, we used bold for the best result, and used italics when two tools perform similarly.

Firstly, we notice that when there is only one condition to solve, no matter the number of iterations, ConFuzz outperforms both state-of-the-art tools. This is due to the fast test case generation, which allows us to quickly explore the loop. Indeed, even though we notice a slight increase in time when the number of iterations grow, it is nothing near KLEE's increase – from 0.805 for 10 iterations to 78.27 for 20 iterations.

When there are 3 or 5 conditions, KLEE outperforms ConFuzz when there are no loop iterations. This makes sense, since solving conditions is what KLEE was made for. However, when the number of loop iteration increases, ConFuzz catches up to KLEE – similar results for 10 iterations – and becomes 150 times better when there are 20 iterations.

As for AFL, its results vary greatly to its non determinism, but it always takes more time than ConFuzz to explore everything. We can however notice that on 20 loop iterations, AFL outperforms KLEE, showing how much KLEE is slowed down by the additional execution paths.

Table 3.1: AFL results

	0 iterations	10 iterations	20 iterations
1 condition	8.337	2.878	3.261
3 conditions	36.18	15.20	25.08
5 conditions	45.05	57.375	66.88

Table 3.2: KLEE results

	0 iterations	10 iterations	20 iterations
1 condition	0.695	0.805	78.27
3 conditions	**0.334**	*1.154*	148.7
5 conditions	**0.573**	*2.385*	305.9

Table 3.3: ConFuzz results

	0 iterations	10 iterations	20 iterations
1 condition	**0.017**	**0.184**	**0.451**
3 conditions	0.968	*1.197*	**1.000**
5 conditions	1.927	*2.142*	**2.152**

Chapter 4

Lightweight Symbolic Execution

Contents

In this chapter we present Lightweight Symbolic Execution, a new version of Dynamic Symbolic Execution. Instead of generating perfect path predicates, Lightweight Symbolic Execution creates approximated path predicates. Under-approximations, to be exact, which we call Easily-Enumerable Path Predicates. We define them as easier to solve, and enumerate solutions for, something we achieve by restraining the expressivity of the constraints. As a result, the path predicates generated by Lightweight Symbolic Execution are correct – any solution of the path predicate is a valid test case for the targeted path – but not complete – not all valid test cases are solutions.

In Section 4.2, we formally define Easily-Enumerable Path Predicates, and introduce the constraint language we use in this thesis, which we prove to be easily-enumerable. We then describe the representation of the execution traces we use (Section 4.3), before presenting the algorithms we use to infer such path predicates on such traces (Section 4.4). Finally, we give some implementation details relevant to our analyses (Section 4.5), before discussing limitations and perspectives (Section 4.6) and related symbolic execution tools (Section 4.7).

Overview

As its name indicates, *Lightweight Symbolic Execution* (LSE) is a variant of Symbolic Execution (SE). Symbolic Execution analyzes a path, be it statically or dynamically, until it reaches its end, then returns a predicate for the path – a constraint on the input variables so that any test case satisfying it will follow the same path. In the case of LSE, we return *Easily-Enumerable Path Predicates*. Expressed with a subset of formulas, they are an approximation of SE's perfect path predicate. Here, we chose to use *under-approximations*, so that the path predicate is *correct*: any solution will follow the trace but not all test cases that follow the trace satisfy the constraint.

We show an example in Figure 4.1 with the program, an execution trace, and both the perfect path predicate and the easily-enumerable one. To make reading easier, we consider that user-defined variables are simply declared in the trace, without concerning ourselves with their exact representation for now. This results in the trace's first instruction: **declare** x, y, z, t, v. In terms of the predicates, they differ because we cannot express difference between variables in our constraint language. Instead, we set the input variables t and v to their concrete value in the execution. As a result, we will always reach past **assert** (c != v), but we will not explore paths with different values for t and v. In exchange, we are able to enumerate solutions using a simple algorithm described in enumTests,

rather than using an SMT solver.

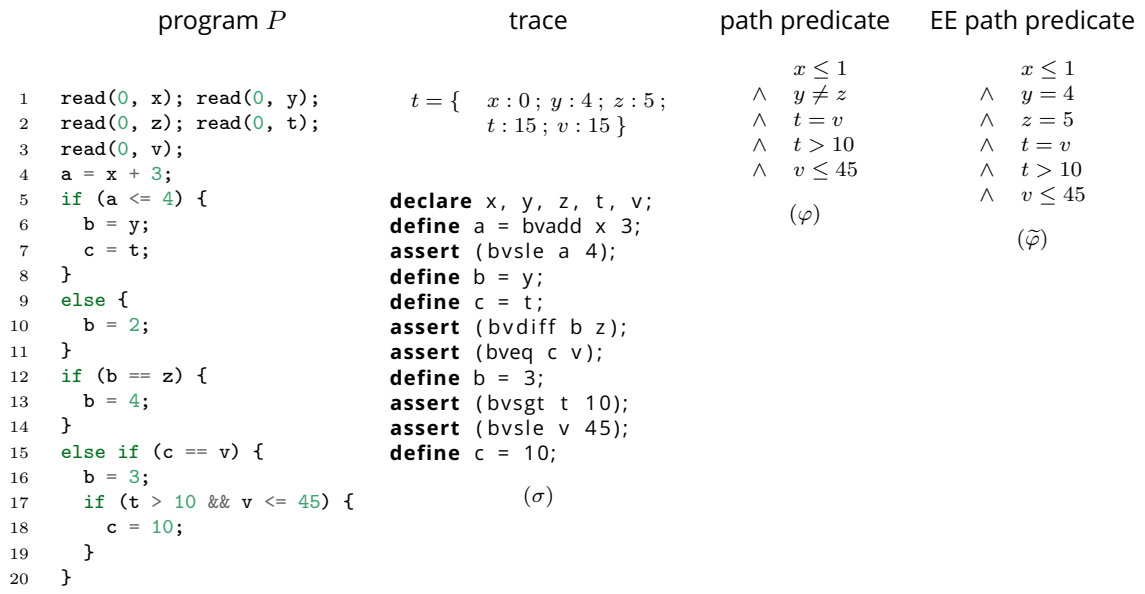


Figure 4.1: Fast-enumerable path predicate

Inverting Mode and Targeting Mode Another difference is that instead of targeting a full path, we target a condition in the path. This allows us to lead the fuzzing past conditions, and then explore below it. To achieve this, we propose two modes for the LSE: targeting and inverting mode, as presented in Section 4.1.

In *targeting mode*, we consider a branch that has just been discovered by an interesting test case – the first transition of yet unexplored code. Our goal is to understand why this test case reached it, to ensure further mutations will continue reaching the new part of the code. We do this by computing the path predicate up to and including the new branch, which we call $\widetilde{\varphi}_t(c)$ – with c the satisfied condition of the branch. We then enumerate solutions, all of which will reach the transition, and explore below. For example, let us consider that the test case t in Figure 4.1 is interesting because it does not satisfy $b = z$. This is illustrated in Figure 4.2a. By sending the path and the interesting transition to the LSE, we would get the path predicate $\widetilde{\varphi}_t(b \neq z) \triangleq x \leq 4 \wedge y = 4 \wedge z = 5$ which targets the new branch $b \neq z$. The constrained fuzzer then enumerate solutions, thus exploring anything below that branch.

In *inverting mode*, the goal is to explore a new part of the code by purposefully taking a yet unexplored branch. This is done by negating a condition and adding it to the path predicate, similarly to Concolic Testing [35]. For example,

for the first one, we would try to negate $c = v$ to invert the condition's output. However, the negated condition is $c \neq v$, which we cannot express in EECL. And since the new condition is a negation of the one in the trace, the current test case is not a solution. As such, we drop this condition. On the other hand, the next condition is $t > 10$. We can negate it, which gives us $t \leq 10$. By adding it to the path predicate which leads to the condition, we get
$\widetilde{\varphi}_i(t > 10) \triangleq x \leq 4 \wedge y = 4 \wedge z = 5 \wedge t = v \wedge t \leq 10$. By negating the last condition on the path, we try creating a path predicate leading to its other branch. The constrained fuzzer then creates a single solution, if one exists, crafting an interesting test case that now explores a new part of the code.

Vocabulary In this section, we differentiate between conditions, constraints and path predicates. *Conditions* are the boolean expressions from the assertions in the trace. There are no guarantees that they are easily-enumerable. *Constraints* are the easily-enumerable translation of a single condition. *Path predicates* – in our case, approximated path predicates – are the conjunction of constraints corresponding to the conditions from the targeted trace. This is what is sent to the constrained fuzzer to be solved.

Defining the Constraint Language

One of the key components of Lightweight Symbolic Execution is its Easily-Enumerable Path Predicates. In this section, we will define the idea of easy-enumerability, and introduce our constraint language.

Easy-enumerability is defined by the complexity of creating at most n different solutions for a given formula.

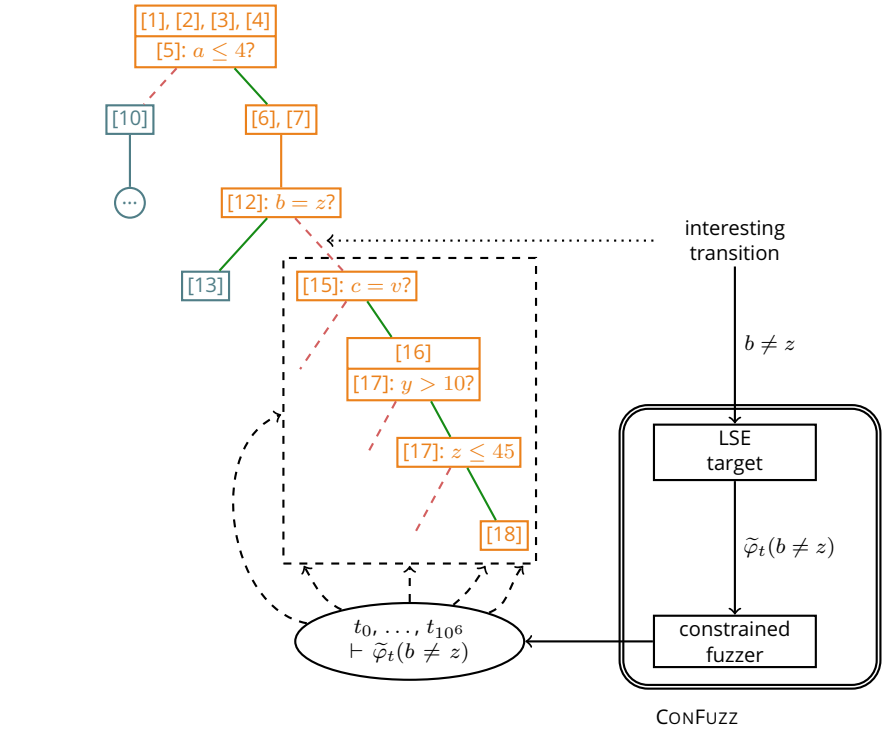
Definition 4.2.1 (Easily-enumerable)**.** *A constraint language CL is* easily-enumerable *if for any formula $\widetilde{\varphi} \in$ CL over a set of input variables X, the complexity of enumerating n solutions (if any) is bounded by $\mathcal{O}(n \times |X|)$.*

The key to having an easily-enumerable constraint language is to restrain its expressivity, in order for the formulas to be easy to solve – thus enumerate – "by design". We propose to do this using the constraint language in Definition 4.2.2.

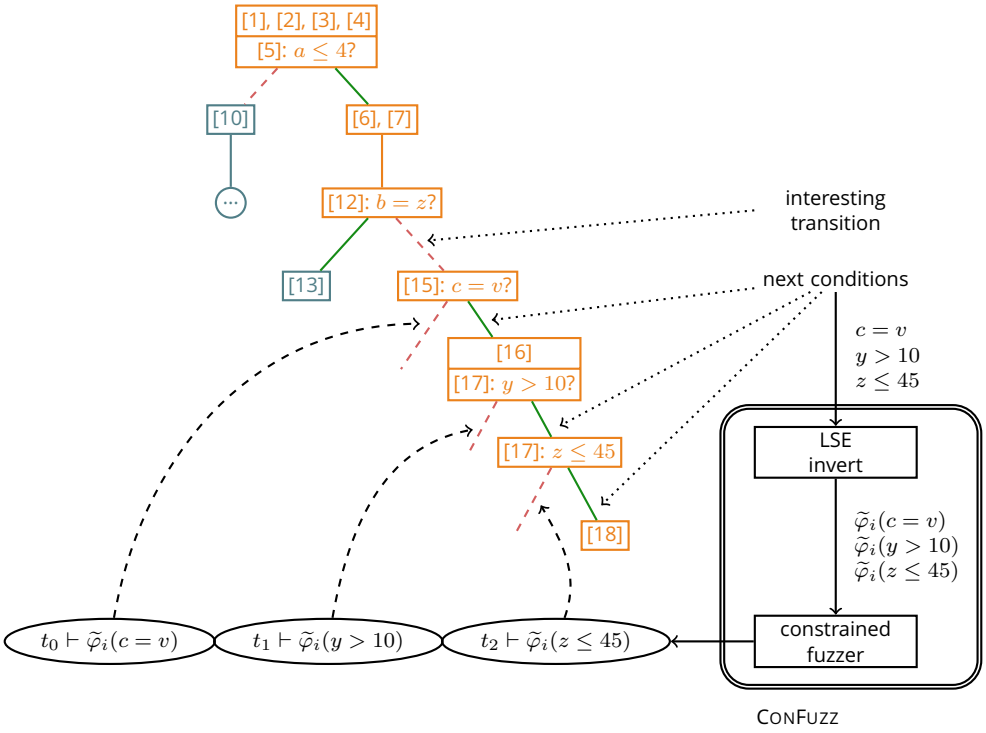
Definition 4.2.2 (Easily-Enumerable Constraint Language (EECL))**.** *Over a set of input variables X, EECL is the language of formulas $\widetilde{\varphi}$ defined by:*

$$\widetilde{\varphi} \triangleq \bigwedge_i x_i \in I_i \wedge \bigwedge_{i,j} x_i = x_j$$

where $x_i, x_j \in X$ and I_i is an integer interval.



ConFuzz - targeting mode



ConFuzz - inverting mode

Figure 4.2: Explanation of ConFuzz's two modes

We also define a normalized version of EECL, NEECL (Definition 4.2.3), which we use to prove the easy-enumerability. Those constraint languages are equivalent, as stated in Theorem 4.2.1.

Definition 4.2.3 (Normalized EECL (NEECL))**.** *Over a set of input variables X, NEECL is the language of formulas $\widetilde{\varphi}^\bullet$ defined by:*

$$\widetilde{\varphi}^\bullet \triangleq \bigwedge_{l \in L} l \in I_l \quad \wedge \bigwedge_{x \in X \setminus L} x = l_x$$

where I_l is an integer interval, $L \subseteq X$ is a subset of the original variables called leaders *and $l_x \in L$.*

Theorem 4.2.1. *Any formula $\widetilde{\varphi} \in$ EECL can be normalized into an equivalent formula $\widetilde{\varphi}^\bullet$ in NEECL.*

Proof. A predicate expressed in EECL contains two types of constraints: variables in an integer interval, and variables equal to variables. Since equality is reflexive ($x = x$), symmetric ($x = y \Leftrightarrow y = x$) and transitive ($x = y \wedge y = z \Rightarrow x = z$), it is an equivalence relation. As a result, sets of variables that are equal form equivalence classes, for each of which we can define a leader l. Translating a formula from EECL to NEECL is a matter of identifying the equivalence classes, adding equality constraints between each member of a class and its leader and merging the constraints on the members to create a single constraint on the leader.

The normalize algorithm describes a way to achieve this, using Union-Find to gather the equivalence classes. Since no information is lost, the solution space of both formulas is the same: $\widetilde{\varphi}^\bullet = \text{normalize}(\widetilde{\varphi})$ and $\widetilde{\varphi}$ are equivalent. Furthermore, since the normalization is only done once, it does not define whether the formula is easily-enumerable. □

We can now prove that formulas expressed using NEECL, and thus EECL, are *easily-enumerable*.

Theorem 4.2.2. *NEECL is easily-enumerable.*

Proof. To prove easy-enumerability, we will first describe an algorithm enumerating solutions, then discuss its complexity. The principle of the algorithm is to set leaders of the class to a value in their interval constraint, one value at a time. Every time we set a leader, we also set the members of the class. Once we have tried every value of a leader, we unset it then backtrack to the last modified variable, until we find a variable with untried values.

Function normalize($\widetilde{\varphi}$, X) pre-processes the formula $\widetilde{\varphi}$

Input: a formula $\widetilde{\varphi}$ and the set of input variables X
Output: $\widetilde{\varphi}^{\bullet}$ a normalized version of the formula $\widetilde{\varphi}$ – or UNSAT

```
1  UF ← create(card(X));
   // Identifying equivalence classes
2  foreach c ∈ φ̃ do
3      if c ≜ x = y then union(UF, x, y);
   // Equality constraints
4  φ̃•_eq ← ⊤;
5  L ← ∅;                                  // the list of class leaders
6  foreach x ∈ X do
7      l ← find(UF, x);
8      if x ≠ l then φ̃•_eq ← φ̃•_eq ∧ (x = l);
9      else L ← append(L, l);
   // Interval constraints
10 φ̃•_c ← ⊤;
11 values ← ∅;                          // constraint computed on a leader
12 foreach c ∈ φ̃ do
13     if c ≜ x ∈ I then
14         l ← find(UF, x);
15         J ← values[l];
           // merging interval on x and interval on leader
16         if I ∩ J ≠ ∅ then values[l] ← I ∩ J;
17         else return UNSAT;
18 foreach l ∈ L do
19     φ̃•_c ← φ̃•_c ∧ l ∈ values[l];
20 return φ̃•_eq ⋀ φ̃•_c
```

To run enumTests, we first breakdown the normalized constraint into a list of the free variables, a list of the leaders, a map associating each leader with its constraint and a map associating each leader to its class members. We use two FILO structures to manage SET and UNSET leaders. The setClass function sets a leader and all member of the associated equivalence class to a value, while the initVars function goes through the UNSET queue and sets every leader, along with its class, to its smallest possible value. Finally, we set unconstrained variables to a random value with setFreeVars. In practice, enumTests will set every leader to its initial value, then increment the last set variable until it has tried all possible values. If this happens, the algorithm unsets it, backtracking until it finds a variable with possible values. The algorithm stops once the values for all leaders have been enumerated, or n tests have been generated.

Complexity-wise, the worst case scenario would be if the algorithm had to backtrack to the top and set every value, every time. Since there are at most $|X|$ variables in the formula, the cost of creating each solution would be bounded by $\mathcal{O}(|X|)$, and the cost of creating n different solutions is bounded by $\mathcal{O}(n \times |X|)$. Thus, even in the worst case scenario, formulas expressed with NEECL are easily-enumerable. □

Procedure setClass(l, c, v) set a leader l and its class c to a value v

Data: t the test case we are creating
Input: l a leader, c its class, v the new value

```
1 t[l] ← v;
2 foreach x ∈ c do
3     t[x] ← v;
```

Procedure initVars(s, u, cs, cl) sets any unset variable to its lowest value

Input: Set the leaders that are set, Unset the leaders that are unset, cstr mapping leaders to their constraint, class mapping leaders to their class

```
1 while Unset ≠ ∅ do
2     l ← pop(Unset);
3     setClass(l, class[l], cstr[l].lo);
4     Set ← push(Set, l);
```

Corollary 4.2.1. *EECL is* easily-enumerable*.*

Procedure setFreeVars(f) sets the free variables to a random value

```
   Input: f the list of free variables
1  foreach x ∈ f do
2      t[x] ← rand();
```

Procedure enumTests(f, L, cstr, class, n) enumerates test cases and runs the program on them

```
   Data: t the test case
   Input: L the leaders, cstr mapping leaders to their constraint, class
          mapping leaders to their class, f the list of free variable, n the
          maximal number of solutions to enumerates
   // Initialisation
1  i ← 0;
2  Set ← ∅;
3  Unset ← L;
4  initVars(Set, Unset, cstr, class);
5  setFreeVars(f);
6  runTarget(t);
   // Enumerating values
7  while Set ≠ ∅ and i < n do
8      l ← pop(Set);
9      if t[l] < cstr[l].hi then
           // we have not tested all values of l
10         setClass(l, class[l], t[l] + 1);
11         initVars(Set, Unset, cstr, class);
12         Set ← push(Set, l);
13         setFreeVars(f);
14         runTarget(t);
15     else
16         Unset ← push(Unset, l);
17     i ← i + 1;
```

Consequences of the approximation

As briefly illustrated in Figure 4.1, by using EECL, we limit the expressivity of the formulas: some conditions cannot be expressed in EECL. We can do one of two things with such conditions: either drop them, or replace them by a constraint expressed in EECL for which the original condition is true. The first one would over-approximate the predicate, as it would allow solutions that do not satisfy the condition. As we want *correct path predicates*, rather than complete, we chose the second option. By over-constraining the condition, we lose possible solutions but only keep valid ones. For example in the example above, by replacing $t \neq v$ with $t = 4 \wedge v = 5$, we lose $\{t \mapsto 2; v \mapsto 3\}$ but solutions will always satisfy all conditions on the path.

The Trace

As Lightweight Symbolic Execution is dynamic, it reasons on an execution trace, obtained by running the Program Under Test on a concrete test case.

Language

The trace we analyze is represented using one of BINSEC's internal languages, called Formula, whose grammar is given in Figures 4.3 to 4.6. It is a simplified version of SMT-LIB, which allows us to use some simplifications [30] from BINSEC on the trace.

A formula is a sequence of entries, which can declare a variable, define a variable as a term, or assert that a condition is true. Terms represent expressions, and can be either a boolean, a bitvector, or an array. Arrays are immutable, which means that every time an array is modified, a new one is created – and usually assigned to a new array variable. In our case, arrays are only used to represent the memory. We distinguish between two arrays: _memory, which represents the program's memory, and _mem_ext, which represents user input. Indeed, unlike the previous examples where we considered input variables, in practice we consider that the input is a table of bytes, accessed by the program through functions such as `read`.

This language is enough to represent a trace, as we do not need jumps: loops are unrolled, and conditional jumps are replaced by assertions on the condition. For example, if we had `if (a <= 4)`, and we know the condition was satisfied for the current test case, we would replace it by **assert** (a <= 4) in the trace, as shown in Figure 4.1.

<trace> → <entry> | <entry> <trace>
<entry> → **declare** <decl>**;**
| **define** <def>**;**
| **assert (**<bl_term>**);**
<decl> → <bv_var> | <ax_var>
<def> → <bl_var> **=** <bl_term>
| <bv_var> **=** <bv_term>
| <ax_var> **=** <ax_term>

Figure 4.3: Trace grammar - instructions

<ax_term> → <ax_var>
| **store** <int> <ax_term> <bv_term> <bv_term>

Figure 4.4: Trace grammar - array terms

<bv_term> → <bv_cst>
| <bv_var>
| <bv_unop> <bv_term>
| <bv_bnop> <bv_term> <bv_term>
| **if** <bl_term> **then** <bv_term> **else** <bv_term>
| **select** <int> <ax_term> <bv_term>
<bv_unop> → **bvnot** | **bvneg**
| **repeat** <int>
| **zero_extend** <int> | **sign_extend** <int>
| **rotate_left** <int> | **rotate_right** <int>
| **extract {**<int>**;** <int>**}**
<bv_bnop> → **bvconcat**
| **bvand** | **bvnand** | **bvor** | **bvnor** | **bvxor** | **bvxnor**
| **bvcmp**
| **bvadd** | **bvsub** | **bvmul** | **bvdiv** | **bvrem**
| **bvshl** | **bvashr** | **bvlshr**

Figure 4.5: Trace grammar - bitvector terms

<*bl_term*> → **true** | **false**
| <*bl_var*>
| <*bl_unop*> <*bl_term*>
| <*bl_bnop*> <*bl_term*> <*bl_term*>
| <*bl_cmp*> <*bl_term*> <*bl_term*>
| <*bv_cmp*> <*bv_term*> <*bv_term*>
| <*ax_cmp*> <*ax_term*> <*ax_term*>
<*bl_unop*> → **blnot**
<*bl_bnop*> → **imply** | **bland** | **blor** | **blxor**
<*bl_cmp*> → **bleq** | **bldiff**
<*bv_cmp*> → **bveq** | **bvdiff**
| **bvult** | **bvule** | **bvugt** | **bvuge**
| **bvslt** | **bvsle** | **bvsgt** | **bvsge**
<*ax_cmp*> → **axeq** | **axdiff**

Figure 4.6: Trace grammar - boolean terms

Specifics

In order to represent the code, we first introduce the idea of undetermined variables. Those are variables that are declared but not defined, and thus do not have a concrete value.

We mostly introduce them when defining stubs. Stubs are used to approximate functions whose code is not available, such as external library calls. We distinguish three kinds of stubs:

user input stub Any function used to get an input from the user needs to be very precisely stubbed, since this information is critical in order to infer constraints on said input. For example, for `read(fd, buf, count)` (Figure 4.7), assuming we know the number of bytes read – info retrieved at runtime, we use it to translate the read into a select from external memory, whose result is stored in the program's memory at the address indicated by the `buf` parameter.

concrete stub In order to optimize the trace, we replace some external library calls by their concrete value. Namely, we do so for allocation functions, which return a memory address. This does not influence the result of the program, but allows to use Read-over-Write simplifications on the trace.

default stub For other functions, for which we do not need a precise stub, we simply consider their return value (eax) to be undetermined. This allows

```
// destination buffer from read call is on the stack
define buf_0 = select <word_size> __mem_ext_0 (bvadd esp_0 8);
// we keep a cursor to the yet unread memory
define tmp_0 = select <read_size> __mem_ext_0 <read_cursor>;
define __memory_1 = store <read_size> __memory_0 buf_0 tmp_0;
define eax_1 = <read_size>;
```

Figure 4.7: Stub for read

us not to lose precision, since we do not constrain them to a single value, and allows us to correctly analyze them – we do not know what they do. On the other hand, this means we ignore any side-effects the functions might have.

Inferring Constraints

The constraint language previously described is the output of our Lightweight Symbolic Execution. Said analysis creates two types of path predicates, as mentioned in Section 4.1:

- $\widetilde{\varphi}_t(c)$ is a path predicate which targets the condition c, meaning we want every condition along the trace to maintain its outcome ;
- $\widetilde{\varphi}_i(c)$ is a path predicate which negates c, meaning we want to target the previous condition, and then add the easily-enumerable constraint associated with $\neg c$.

Given $[\![c]\!]_{EECL}$ the constraint inferred for the condition c, we can formally define both (Definition 4.4.1).

Definition 4.4.1.

$$\begin{aligned} \widetilde{\varphi}_t(c_n) &\triangleq \textstyle\bigwedge_{l=0}^{n} [\![c_l]\!]_{EECL} \\ \widetilde{\varphi}_i(c_n) &\triangleq \widetilde{\varphi}_t(c_{n-1}) \wedge [\![\neg c_n]\!]_{EECL} \end{aligned}$$

The difficulty in inferring an EECL path predicate is computing the individual $[\![c]\!]_{EECL}$ constraint for each condition. We do so with different analyses, which are *dynamic* and *backward*. This means they are based on a concrete execution trace, and start at the condition, going backward on relevant instructions.

Assumptions and notations We use two ways to describe the algorithms: inference rules and pseudo-OCaml, and will use examples to illustrate how the algorithms work. Following those two representations, the elements of the

trace are represented either directly as they are in the grammar – <*bv_expr*> $=_{bv}$ <*bv_expr*> – or as an algebraic type which follows the grammar – BvCmp(BvEqual, BvExpr e_1, BvExpr e_2).

During the analyses, we consider that a first pass over the trace gave us two different structures:

- a map var_{def} of type $bv\_var \mapsto bv\_term$, so that $var_{def}[x] = e$ iff **define** x = e is in the trace ;
- a map mem_{def} of type $(int,\ ax\_var,\ bv\_term) \mapsto bv\_term$, so that $mem_{def}[size,\ m_k,\ idx] = elt$ iff the last modification of the memory at idx was **define** m_(l+1) = store (size, m_l, idx, elt), with $l < k$.

Finally, for any given condition $cond$, we want to compute the corresponding constraint in EECL, if it exists. We write this as $cond \rightsquigarrow c$, where $c = [\![cond]\!]_{EECL}$. In particular, $e \in [\![a; b]\!] \rightsquigarrow c$ means that the condition $e \in [\![a; b]\!]$ is resolved by the constraint c, expressed in EECL. For example, in Section 4.1, we mentioned $\widetilde{\varphi}_t(b \neq z)$, which we compute as described in Figure 4.8.

$$\begin{array}{rcl} \widetilde{\varphi}_t(b \neq z) & \triangleq & [\![a \leq 4]\!]_{EECL} \wedge [\![b \neq z]\!]_{EECL} \\ [\![a \leq 4]\!]_{EECL} & = & x \in [\![min; 1]\!] \\ [\![b \neq z]\!]_{EECL} & = & y \in [\![4; 4]\!] \wedge z \in [\![5; 5]\!] \end{array}$$

Result: $\widetilde{\varphi}_t(b \neq z) \triangleq x \in [\![min; 1]\!] \wedge y \in [\![4; 4]\!] \wedge z \in [\![5; 5]\!]$

Figure 4.8: Computing $\widetilde{\varphi}_t(b \neq z)$

If we cannot compute a constraint, we return $\top$, which represents the empty constraint.

Orchestration

In order to compute the constraints we use one of three analyses:

- the equality analysis (Section 4.4.2) analyzes (**bveq** <*bv_term*> <*bv_term*>) conditions, and determines if they are of the form $mem\_ext[i..j] = mem\_ext[k..l]$;
- the value analysis (Section 4.4.3) analyzes (<*bv_cmp*> <*bv_term*> <*bv_cst*>) conditions (except for difference comparisons), and determines if they are of the form $mem\_ext[i..j] \in I$;

- the dependency analysis (Section 4.4.4) analyzes any type of conditions, to determine which, if any, bytes of the input the condition depends on.

The analyzeCond function determines if we can use equality or value analysis, depending on the condition. This is done with a pattern-matching on the condition, which is a <*bl_term*>. If it is not either of the two patterns, or if the analyses are not conclusive, we run the dependency analysis. We also use the auxiliary function negateCond, which modified the operators of a negated condition – for example make $\neg(a = b)$ into $a \neq b$.

Equality Analysis

If the condition $cond$ is of the form **bveq** <*bv_term*> <*bv_term*> – assuming that neither expression is a constant – we will do an alias analysis, backtracking along the expression definitions, as described in the functions alias and analyzeEquality. If both terms end up being selections from mem_ext, we have the constraint in EECL: $mem\_ext[i, n] = mem\_ext[j, n]$ – meaning the n bytes starting from i are equal to the n bytes starting from j. To be noted, equal expressions are expected to be of the same size, hence the identical n value. Otherwise, we conclude that the condition cannot be expressed with an equality constraint.

Function analyzeCond($cond$, $invMode$) determines which analysis to run on $cond$, depending on whether it was negated

Input: $cond$, a boolean term from the trace ; $invMode$, a boolean indicating whether we negated the condition
Output: $cstr$, an EECL constraint, possibly empty

```
1  switch cond do
2      case BlTrue or BlFalse do
3          return ⊤;
4      case BlVar(v) do
5          return analyzeCond(var_def[v], invMode);
6      case BlUnop(BlNot, cond') do
7          invCond ← negateCond(cond');
8          return analyzeCond(invCond, invMode);
9      case BlBnop(BlAnd, cond_1, cond_2) do
10         cstr_1 ←analyzeCond(cond_1, invMode);
11         cstr_2 ←analyzeCond(cond_2, invMode);
12         return cstr_1 ∧ cstr_2;
13     case BvCmp(op, bv, BvCst(c)) do
14         if op is BvDiff then
15             return analyzeDependencies(cond, invMode);
16         else
17             cstr ← analyzeValues(op, bv, c);
18             if cstr = ⊤ then
19                 return analyzeDependencies(cond, invMode);
20             else
21                 return cstr;
22     case BvCmp(_, BvCst(_), _) do
           // we negate the condition to exchange the variable and
              the constant
23         return analyzeCond(negateCond(cond), invMode);
24     case BvCmp(BvEqual, bv_1, bv_2) do
25         cstr ← analyzeEquality(bv_1, bv_2);
26         if cstr = ⊤ then
27             return analyzeDependencies(cond, invMode);
28         else
29             return cstr;
30     otherwise do
31         return analyzeDependencies(cond, invMode);
```

Function alias(bv) determines whether the term bv is from the input

Input: bv, the term we want to analyze
Output: Some(*idx, n*), where idx is the index of the input selection and n its size, or None

```
1  switch bv do
2      case BvVar(v) do
3          if v ∈ var_def then
4              return alias(var_def[v]);
5          else
6              return None
7      case Select(n, mem, idx) do
8          if mem is mem_ext then
9              return Some(idx, n);
10         else
11             return alias(mem_def[n, mem, idx])
12     otherwise do
13         return None
```

Function analyzeEquality(bv_1, bv_2) applies equality analysis to $bv_1 = bv_2$

Input: bv_1 and bv_2, two terms which are constrained to be equal
Output: $cstr$, an EECL constraint, possibly empty

```
1  alias_1, alias_2 ← alias(bv_1), alias(bv_2);
2  switch alias_1, alias_2 do
3      case Some(i, n), Some(j, m) do
4          assert(n = m);
5          return (i, n) = (j, n);
6      otherwise do return ⊤;
```

Value Analysis

When the condition is of the form <*bv_cmp*> <*bv_term*> <*bv_cst*>, there is a chance it might correspond to an interval constraint for bits from the input. To analyze such a condition, we first translate it into a constraint of the form $e \in [\![a; b]\!]$ as described in analyzeValues.

Since we work with bitvector terms their size and sign give us a general interval constraint: $[-2\hat{}31; 2\hat{}31 - 1]$ if it is signed, $[0; 2\hat{}32 - 1]$ if it is unsigned. In the case of analyzeValues, we get the size of the term using an auxiliary `sizeof` function, and we deduce the sign from the comparison operator used. We then restrict this interval based on the condition.

Once we have an interval constraint, the function values backtracks on the term by following the rules described in Figures 4.9 to 4.12, modifying the interval as we go.

In our analysis, we do not consider arithmetic overflows. Technically, if we have $x+3 \in [\![min; 4]\!]$ then $x \in [\![min; 1]\!] \cup [\![max-2; max]\!]$, because $(max-2)+3 = min$. However, since we cannot express unions of intervals, we restrain the constraint to $[\![min; 1]\!]$.

Function analyzeValues(op, bv, cst) applies value analysis to the condition $bv\ op\ cst$

Input: op the operator, bv the bitvector term and cst the constant for the condition

Output: $cstr$, an EECL constraint, possibly empty

```
1  size ← sizeof(bv);
2  switch op do
3      case BvEqual do
4          return values(bv ∈ [[cst; cst]])
5      case BvSle do
6          return values(bv ∈ [[signedMin(size); cst]])
7      case BvSlt do
8          return values(bv ∈ [[signedMin(size); cst − 1]])
9      case BvSge do
10         return values(bv ∈ [[cst; signedMax(size)]])
11     case BvSgt do
12         return values(bv ∈ [[cst + 1; signedMax(size)]])
13     case BvUle do
14         return values(bv ∈ [[0; cst]])
15     case BvUlt do
16         return values(bv ∈ [[0; cst − 1]])
17     case BvUge do
18         return values(bv ∈ [[cst; unsignedMax(size)]])
19     case BvUgt do
20         return values(bv ∈ [[cst + 1; unsignedMax(size)]])
21     otherwise do return ⊤;
```

Initial rules (Figure 4.9) If the condition is on a constant, there is no constraint: with $42 \in [\![40; 50]\!]$, no user input is involved. Similarly, if the condition is on a nondet variable, we cannot infer any constraint on the user input.

On the other hand, if the condition is on a select from mem_ext, we have found a constraint on user input, which directly translates into an EECL constraint. To identify the constrained bytes, we need the concrete value of the <*bv_term*> that indicates the index. We do so using an evaluation function which we will describe in Section 4.4.4, and which we represent with $\Downarrow$. Since the size is already an integer, we do not need to evaluate it. However, we want to ensure that we will always access the same bytes of the input. To do so, we infer the constraint for bv_{idx} to be equal to its evaluated value v_{idx}, and add this to the resulting constraint.

Finally, if the condition is on a variable (one that is not nondet), we simply backtrack on the variable's definition, which is stored in var_{def} And if we find a selection from the program's memory, we get the element using mem_{def}, while also computing the condition for the index to be equal to its current value.

$$\frac{}{\texttt{cst} \in [\![a; b]\!] \rightsquigarrow \top}\ \text{CONSTANT} \qquad \frac{x \notin var_{def}}{x \in [\![a; b]\!] \rightsquigarrow \top}\ \text{NONDET}$$

$$\frac{bv_{idx} \Downarrow v_{idx} \qquad c_{idx} = \text{analyzeCond}(\textbf{bveq}\ bv_{idx}\ v_{idx})}{\textbf{select}\ n\ \textbf{\_\_mem\_ext}\ bv_{idx} \in [\![a; b]\!] \rightsquigarrow c_{idx} \wedge (v_{idx}, n) \in [\![a; b]\!]}\ \text{INPUT}$$

$$\frac{var_{def}[x] = bv \qquad bv \in [\![a; b]\!] \rightsquigarrow c}{x \in [\![a; b]\!] \rightsquigarrow c}\ \text{VAR}$$

$$\frac{bv_{idx} \Downarrow v_{idx} \qquad mem_{def}[ax_{mem}, v_{idx}, n] = bv_{elt} \qquad bv_{elt} \in [\![a; b]\!] \rightsquigarrow c_{elt}}{\textbf{select}\ n\ ax_{mem}\ bv_{idx} \in [\![a; b]\!] \rightsquigarrow \text{analyzeCond}(\textbf{bveq}\ bv_{idx}\ v_{idx}) \wedge c_{elt}}\ \text{SELECT}$$

Figure 4.9: Value analysis - Initial rules

Simple rules (Figure 4.10) With some operations, it is possible to straightforwardly propagate the operation to the interval. For example, mathematics tell us that $a < -e < b$ is equivalent to $-b < e < -a$. As such, we can backtrack on $-e \in [\![a; b]\!]$ by inferring the constraint for $e \in [\![-b; -a]\!]$. Another example is that if we have $e + cst \in [\![a; b]\!]$ with cst a constant, then we can infer the constraint for $e \in [\![a - cst; b - cst]\!]$. And if we are not adding a constant, we return the empty constraint.

$$\frac{bv \in [\![-b; -a]\!] \rightsquigarrow c}{\textbf{bvneg}\ bv \in [\![a; b]\!] \rightsquigarrow c}\ \text{NEG}$$

$$\frac{bv \in [\![a - \texttt{cst}; b - \texttt{cst}]\!] \rightsquigarrow c}{\textbf{bvadd}\ bv\ \texttt{cst} \in [\![a; b]\!] \rightsquigarrow c}\ \text{ADD CST} \qquad \frac{}{\textbf{bvadd}\ bv_1\ bv_2 \in [\![a; b]\!] \rightsquigarrow \top}\ \text{ADD DEF}$$

$$\frac{bv \in [\![a + \texttt{cst}; b + \texttt{cst}]\!] \rightsquigarrow c}{\textbf{bvsub}\ bv\ \texttt{cst} \in [\![a; b]\!] \rightsquigarrow c}\ \text{SUB CST 1} \qquad \frac{bv \in [\![\texttt{cst} - b; \texttt{cst} - a]\!] \rightsquigarrow c}{\textbf{bvsub}\ \texttt{cst}\ bv \in [\![a; b]\!] \rightsquigarrow c}\ \text{SUB CST 2}$$

$$\frac{}{bv_1 - bv_2 \in [\![a; b]\!] \rightsquigarrow c}\ \text{SUB DEF}$$

Figure 4.10: Value analysis - Simple rules

Equality rules (Figure 4.11) Some of the rules can only be applied when the term is constrained to a single value, meaning we are analyzing an equality. For example, if $!a = 4$, we know that $a = !4$. On the other hand, we cannot infer anything from $!a < 4$, as the boolean not operation is not symmetric wrt inequations, and will then return the empty constraint.

$$\frac{bv \in [\![!a; !a]\!] \rightsquigarrow c}{\textbf{bvnot}\ bv \in [\![a; a]\!] \rightsquigarrow c}\ \text{NOT EQ} \qquad \frac{}{\textbf{bvnot}\ bv \in [\![a; b]\!] \rightsquigarrow \top}\ \text{NOT DEF}$$

$$\frac{bv \in [\![a \oplus \texttt{cst}; a \oplus \texttt{cst}]\!] \rightsquigarrow c}{\textbf{bvxor}\ bv\ \texttt{cst} \in [\![a; a]\!] \rightsquigarrow c}\ \text{XOR EQ} \qquad \frac{}{\textbf{bvxor}\ bv_1\ bv_2 \in [\![a; b]\!] \rightsquigarrow \top}\ \text{XOR DEF}$$

Figure 4.11: Value analysis - Equality rules

Complex rules (Figure 4.12) In order to analyze more complex operations, we introduce the idea of "constrained bits": $e\{lo; hi\} \in [\![a; b]\!]$ means only the bits from lo to hi of e are constrained to $[\![a; b]\!]$. If no bits are indicated, it means the whole expression is constrained. We use this for the following rules:

extension a constraint on an extended term can only be propagated to the term if the constraint does not concern the extension. For example, if $(\textbf{sign\_extend}\ 8\ bv)\{0; 15\} \in [\![a; b]\!]$, where bv is of size 24, then we can backtrack to $bv\{0; 15\} \in [\![a; b]\!]$. Otherwise, we will consider we cannot conclude and return the empty constraint.

extraction we can propagate a constraint on an extraction to the original term by offsetting the constrained bits. For example, if $(\textbf{extract}\ \{8; 23\}\ bv)\{0; 7\} \in$

$[\![a; b]\!]$, the first 8 bits of the extraction are bits 8 to 15 of e: we backtrack on $bv\{8; 15\} \in [\![a; b]\!]$.

concatenation there are three cases when constraining the result of a concatenation:

- if the constraint applies only to one of the terms, we propagate the constraint on this term. (**bvconcat** bv_h $bv_l)\{8; 15\} \in [\![a; b]\!]$, when `sizeof`$(bv_l) = 8$, means $bv_h\{0; 7\} \in [\![a; b]\!]$;
- if the constraint is an equality, we can just split it: (**bvconcat** $bv_l\, bv_h = a \Leftrightarrow bv_l = lo(a) \wedge bv_h = hi(a)$, where $lo(a)$ is the lower part of a, from 0 to `sizeof`$(bv_l) - 1$, and $hi(a)$ is the rest ;
- by default, we return an empty constraint.

comparison bvcmp is an operation which compares two bitvectors and returns the constant 1 if they are equal. As such, if the result of the comparison is constrained to 1, we need the compared bitvectors to be equal. This gives us a new condition, on which we call analyzeCond. On the other hand, if the result is equal to zero, we ignore the condition since it would be a difference.

if then else if we are constraining the result of an "if then else" term, we can use the information from the constraint to determine whether the condition is verified. For example, if we have (**if** $cond$ **then** 4 **else** $42) \in [\![0; 5]\!]$, then we know $cond$ was verified, and can start a new analysis on $cond$. Thus, by evaluating the concrete values of both branches results, we can sometimes infer information about the condition, and analyze it. If not, we just return the empty constraint.

Default rules There are operations for which it is never possible to update the interval and backtrack, for example **bvnand**. For those operations, who do not have a formally defined inference rules, we just return the empty constraint.

$$\frac{hi < \texttt{size}(bv) \qquad bv\{lo; hi\} \in [\![a; b]\!] \rightsquigarrow c}{(\textbf{zero\_extend}\ n\ bv)\{lo; hi\} \in [\![a; b]\!] \rightsquigarrow c}\ \text{ZEXT} \qquad \frac{}{(\textbf{zero\_extend}\ n\ bv) \in [\![a; b]\!] \rightsquigarrow \top}\ \text{ZEXT DEF}$$

$$\frac{hi < \texttt{size}(bv) \qquad bv\{lo; hi\} \in [\![a; b]\!] \rightsquigarrow c}{(\textbf{sign\_extend}\ n\ bv)\{lo; hi\} \in [\![a; b]\!] \rightsquigarrow c}\ \text{SEXT} \qquad \frac{}{(\textbf{sign\_extend}\ n\ bv) \in [\![a; b]\!] \rightsquigarrow \top}\ \text{SEXT DEF}$$

$$\frac{bv\{lo + l; hi + l\} \in [\![a; b]\!] \rightsquigarrow c}{(\textbf{extract}\ \{l; h\}\ bv)\{lo; hi\} \in [\![a; b]\!] \rightsquigarrow c}\ \text{EXTRACT}$$

$$\frac{hi < \texttt{sizeof}(bv_l) \qquad bv_l\{lo; hi\} \in [\![a; b]\!] \rightsquigarrow c}{(\textbf{bvconcat}\ bv_l\ bv_h)\{lo; hi\} \in [\![a; b]\!] \rightsquigarrow c}\ \text{CONCAT LOW}$$

$$\frac{lo \geq \texttt{sizeof}(bv_l) \qquad bv_h\{lo - \texttt{sizeof}(bv_l); hi - \texttt{sizeof}(bv_l)\} \in [\![a; b]\!] \rightsquigarrow c}{(\textbf{bvconcat}\ bv_l\ bv_h)\{lo; hi\} \in [\![a; b]\!] \rightsquigarrow c}\ \text{CONCAT HIGH}$$

$$\frac{bv_l \in [\![\texttt{lo}(a); \texttt{lo}(a)]\!] \rightsquigarrow c \qquad bv_h \in [\![\texttt{hi}(a); \texttt{hi}(a)]\!] \rightsquigarrow d}{(\textbf{bvconcat}\ bv_l\ bv_h) \in [\![a; a]\!] \rightsquigarrow c \wedge d}\ \text{CONCAT EQ}$$

$$\frac{lo < \texttt{size}(bv_l) < hi}{(\textbf{bvconcat}\ bv_l\ bv_h)\{lo; hi\} \in [\![a; b]\!] \rightsquigarrow \top}\ \text{CONCAT DEF}$$

$$\frac{c = \text{analyzeCond}(\textbf{bveq}\ bv_1\ bv_2)}{(\textbf{bvcmp}\ bv_1\ bv_2) \in [\![1; 1]\!] \rightsquigarrow c}\ \text{CMP EQ ONE} \qquad \frac{}{(\textbf{bvcmp}\ bv_1\ bv_2) \in [\![a; b]\!] \rightsquigarrow \top}\ \text{CMP DEF}$$

$$\frac{bv_t \Downarrow v_t \quad bv_e \Downarrow v_e \qquad v_t \in [\![a; b]\!] \wedge v_e \notin [\![a; b]\!] \qquad c = \text{analyzeCond}(bl_c)}{(\textbf{if}\ bl_c\ \textbf{then}\ bv_t\ \textbf{else}\ bv_e) \in [\![a; b]\!] \rightsquigarrow c}\ \text{ITE THEN}$$

$$\frac{bv_t \Downarrow v_t \quad bv_e \Downarrow v_e \qquad v_t \notin [\![a; b]\!] \wedge v_e \in [\![a; b]\!] \qquad c = \text{analyzeCond}(\textbf{blnot}\ bl_c)}{(\textbf{if}\ bl_c\ \textbf{then}\ bv_t\ \textbf{else}\ bv_e) \in [\![a; b]\!] \rightsquigarrow c}\ \text{ITE ELSE}$$

$$\frac{}{(\textbf{if}\ bl_c\ \textbf{then}\ bv_t\ \textbf{else}\ bv_e) \in [\![a; b]\!] \rightsquigarrow \top}\ \text{ITE DEF}$$

Figure 4.12: Value analysis - Complex rules

Dependency Analysis

When it is impossible to compute a constraint directly from the condition, we fall back on a dependency analysis. This only applies in targeting mode, as we are trying to recreate the trace we had when executing the PUT on the interesting test case, which we call t_i. By analyzing the input dependencies of the condition, we can identify which n input bytes, starting at idx, it depends on. And since we always have a valuation of the variable for which the condition is satisfied – that of t_i, we only need to add a constraint $\bigwedge_{k=idx}^{n} k \in [\![t_i[k]; t_i[k]]\!]$. This does not work when negating a condition, as we then do not have any valuation for which the condition is satisfied. As a result, the function is called with the condition and a boolean indicating whether it was negated, and returns $\top$ if the boolean is true.

When analyzing the trace, we consider two types of dependency information:

- byte-level dependencies, which we represent with an array of sets, give the dependencies for each byte of the term ;
- term-level dependencies, which we represent with a set, give the overall dependencies of the term.

In this analysis, we compute both dependencies and values, as we need the latter to compute dependencies in some cases. This is also the function used to compute index values during value analysis. Values are either a constant or a symbolic variable associated to an offset. This allows us to deduce information even when we do not have the concrete value, for example resolving a select and a store with the same symbolic index. The operations on values are as follows:

- if all operands are constants, apply the operation ;
- if the operation is either sym + cst, sym - cst or cst + sym, update the offset of the symbolic variable accordingly ;
- otherwise create a new symbolic variable.

To compute the dependencies, we define a `merge` function which takes the dependencies of two terms and:

- if both are byte-level, merges the dependencies for each byte ;
- if one is byte-level and the other is term-level, and the dependencies of the second ones to each byte of the first one ;

- if both are term-level, compute the union.

Meanwhile, the `squash` function will compute the union of all dependencies it was given, on a term-level. Examples for both are shown in Figure 4.13.

$$\begin{aligned}
\texttt{merge}\ [\{0\}\{\}\{1\}]\ [\{3\}\{2\}\{\}] &= [\{0;3\}\{2\}\{1\}] \\
\texttt{merge}\ [\{0\}\{\}\{1\}]\ \{2;3\} &= [\{0;2;3\}\{2;3\}\{1;2;3\}] \\
\texttt{merge}\ \{0;1\}\ \{2;3\} &= \{0;1;2;3\} \\
\\
\texttt{squash}\ [\{0\}\{\}\{1\}] &= \{0;1\} \\
\texttt{squash}\ [\{0\}\{\}\{1\}]\ \{2;3\} &= \{0;1;2;3\}
\end{aligned}$$

Figure 4.13: Merge and squash examples

Initial rules (Figure 4.14) Constants and nondet variables do not depend on the input: we just create an empty byte-level dependency, of the size of the term. On the other hand, when the term is a selection from mem_ext, it directly depends on the user input, as well as the selected index: we compute the index's dependencies, initialize the element's dependencies to the selected bytes, and merge the two. As for variables and selects from memory, as for the previous analyses we look up the element in the definition table.

Byte-level operations (Figure 4.15) preserve byte-level dependencies. In some cases, we just merge the dependencies of the two bitvector terms involved in the operation. For example, with bit-wise binary operations, the k^{th} byte of the result directly depends on the k^{th} byte of each operand: we just merge the dependencies. This also applies to boolean operators, which take two single-byte terms and also return a single byte.

$$bv_1 \Downarrow [\{0\}\{1\}] \,\&\, bv_2 \Downarrow [\{4\}\{5\}] \Rightarrow \textbf{bvand}\ bv_1\ bv_2 \Downarrow [\{0;4\}\{1;5\}]$$

Some unary operations preserve the byte-level aspect of the dependencies, but modify them. For example, if there is an extraction, we only keep the bytes from the term which were extracted.

$$bv \Downarrow [\{0\}\{1\}] \Rightarrow \textbf{bvextract}\ \{8;15\}\ bv \Downarrow [\{1\}]$$

For those operations, if the dependency is term-level instead, we just keep the dependency as is. Finally, if we are concatenating two terms, there are 3 cases:

- if the dependencies of both terms are byte-level, we just concatenate them:

$$bv_l \Downarrow [\{0\}\{1\}] \,\&\, bv_h \Downarrow [\{4\}\{5\}] \Rightarrow \textbf{bvconcat}\ bv_l\ b_h \Downarrow [\{0\}\{1\}\{4\}\{5\}]$$

- if one is term-level, we translate it into byte-level by duplicating the dependency, which by definition applies to all bytes, then concatenate the dependencies:

$$bv_l \Downarrow \{0;1\} \,\&\, bv_h \Downarrow [\{4\}\{5\}] \Rightarrow \textbf{bvconcat}\ bv_l\ b_h \Downarrow [\{0;1\}\{0;1\}\{4\}\{5\}]$$

- if both are term-level, we translate them into byte-level and concatenate:

$$bv_l \Downarrow \{0;1\} \,\&\, bv_h \Downarrow \{4;5\} \Rightarrow \textbf{bvconcat}\ bv_l\ b_h \Downarrow [\{0;1\}\{0;1\}\{4;5\}\{4;5\}]$$

Other operations (Figure 4.16) For operations that do not preserve byte-level dependencies, the analysis just squashes the dependencies of all operands, so as to get the overall dependency of the result. For example, if we are adding two terms, the possibility of carry means any byte can depend on other bytes: it is simpler to consider that all of the result inherits the operands' dependency.

$$bv_1 \Downarrow [\{0\}\{1\}] \,\&\, bv_2 \Downarrow \{4;5\} \Rightarrow \textbf{bvadd}\ bv_1\ bv_2 \Downarrow \{0;1;4;5\}$$

For if then else operations, we might be able to use the values of the terms to refine the analysis:

- if we have **if** bl_c **then** bv_t **else** bv_e, and we can evaluate bl_c to the concrete value of 1 (resp 0), we know the result of the operation is bv_t (resp bv_e), and we analyze it. We also squash the result with the dependencies of the boolean term, since it is determinant.

- otherwise, we do not know which term is the result: we squash the dependencies of all three terms, and return a new symbolic variable, nondet, as the value.

$$\frac{}{\texttt{cst} \Downarrow [\ \underbrace{\{\}}_{\text{size of } \texttt{cst} \text{ times}}\ ],\ \texttt{cst}}\ \text{CST} \qquad \frac{v \notin var_{def}}{v \Downarrow [\ \underbrace{\{\}}_{\text{size of v times}}\ ],\ v+0}\ \text{NONDET}$$

$$\frac{}{\textbf{true} \Downarrow [\{\}],\ 1}\ \text{TRUE} \qquad \frac{}{\textbf{false} \Downarrow [\{\}],\ 0}\ \text{FALSE}$$

$$\frac{bv_{idx} \Downarrow d_{idx}, v_{idx} \qquad d = \texttt{merge}\ (\texttt{squash}\ d_{idx})\ [\{v_{idx}\} \ldots \{v_{idx}+n-1\}]}{\textbf{select}\ n\ \textbf{\_\_mem\_ext}\ bv_{idx} \Downarrow d,\ input[v_{idx} \ldots (v_{idx}+n-1)]}\ \text{INPUT}$$

$$\frac{var_{def}[x] = bv \qquad bv \Downarrow d, v}{x \Downarrow d,\ v}\ \text{VAR}$$

$$\frac{bv_{idx} \Downarrow d_{idx}, v_{idx} \qquad mem[ax_{mem}, v_{idx}, n] = bv_{elt} \qquad bv_{elt} \Downarrow d_{elt}, v_{elt}}{\textbf{select}\ n\ ax_{mem}\ bv_{idx} \Downarrow \texttt{merge}\ (\texttt{squash}\ d_{idx})\ d_{elt}, v_{elt}}\ \text{SELECT}$$

Figure 4.14: Initial dependencies

$$\frac{bl \Downarrow d, v}{\textbf{blnot}\ bl \Downarrow d, !v}\ \text{BLNOT} \qquad \frac{bv \Downarrow d, v}{\textbf{bvnot}\ bv \Downarrow d, !v}\ \text{BVNOT}$$

$$\frac{bl_1 \Downarrow d_1, v_1 \qquad bl_2 \Downarrow d_2, v_2}{(\diamond_{bl} \in \textbf{bleq} \mid \textbf{bldiff})\ bl_1\ bl_2 \Downarrow \texttt{merge}\ d_1\ d_2, v_1 \diamond v_2}\ \text{BLCMP}$$

$$\frac{bl_1 \Downarrow d_1, v_1 \qquad bl_2 \Downarrow d_2, v_2}{(\diamond_{bl} \in \textbf{imply} \mid \textbf{bland} \mid \textbf{blor} \mid \textbf{blxor})\ bl_1\ bl_2 \Downarrow \texttt{merge}\ d_1\ d_2, v_1 \diamond v_2}\ \text{BLBNOP}$$

$$\frac{bv_1 \Downarrow d_1, v_1 \qquad bv_2 \Downarrow d_2, v_2}{(\diamond_{bv} \in \textbf{bvand} \mid \textbf{bvnand} \mid \textbf{bvor} \mid \textbf{bvnor} \mid \textbf{bvxor} \mid \textbf{bvxnor})\ bv_1\ bv_2 \Downarrow \texttt{merge}\ d_1\ d_2, v_1 \diamond v_2}\ \text{BVBNOP BITWISE}$$

$$\frac{bv \Downarrow [\{d^0\} \ldots \{d^k\}], v}{\textbf{extract}\ \{lo; hi\}\ bv \Downarrow [\{d^{\frac{lo}{8}}\} \ldots \{d^{\frac{hi}{8}}\}], \texttt{extract}\ \{lo; hi\}\ v}\ \text{EXTRACT}$$

$$\frac{bv \Downarrow [\{d^0\} \ldots \{d^k\}], v}{\textbf{repeat}\ n\ bv \Downarrow [\underbrace{\{d^0\} \ldots \{d^k\}}_{\frac{n}{8}\ \text{times}}], \texttt{repeat}\ n\ v}\ \text{REPEAT}$$

$$\frac{bv \Downarrow [\{d^0\} \ldots \{d^k\}], v}{\textbf{zero\_extend}\ n\ bv \Downarrow [\{d^0\} \ldots \{d^k\}\ \underbrace{\{\}}_{\frac{n}{8}\ \text{times}}], \texttt{zero\_extend}\ n\ v}\ \text{ZEXT}$$

$$\frac{bv \Downarrow [\{d^0\} \ldots \{d^k\}], v}{\textbf{sign\_extend}\ n\ bv \Downarrow [\{d^0\} \ldots \{d^k\}\ \underbrace{\{d^k\}}_{n\ \text{times}}], \texttt{sign\_extend}\ n\ v}\ \text{SEXT}$$

$$\frac{bv \Downarrow \{\ldots\}, v}{(\diamond_{bv} \in \textbf{extract} \mid \textbf{repeat} \mid \textbf{zero\_extend} \mid \textbf{sign\_extend})\ k\ bv \Downarrow \{\ldots\}, \diamond\ k\ v}\ \text{DEFAULT BVUNOP}$$

$$\frac{bv_l \Downarrow [\{d_l^0\} \ldots \{d_l^k\}], v_l \qquad bv_h \Downarrow [\{d_h^0\} \ldots \{d_h^j\}], v_h}{\textbf{bvconcat}\ bv_l\ bv_h \Downarrow [\{d_l^0\} \ldots \{d_l^k\}\{d_h^0\} \ldots \{d_h^j\}], \texttt{concat}\ v_l\ v_h}\ \text{CONCAT BYTE AND BYTE}$$

$$\frac{bv_l \Downarrow \{d_l^0, \ldots, d_l^k\}, v_l \qquad bv_h \Downarrow [\{d_h^0\} \ldots \{d_h^j\}], v_h}{\textbf{bvconcat}\ bv_l\ bv_h \Downarrow [\underbrace{\{d_l^0, \ldots, d_l^k\}}_{\text{size of } bv_l\ \text{times}}\{d_h^0\} \ldots \{d_h^j\}], \texttt{concat}\ v_l\ v_h}\ \text{CONCAT BYTE AND TERM}$$

$$\frac{bv_l \Downarrow \{d_l^0 \ldots d_l^k\}, v_l \qquad bv_l \Downarrow \{d_h^0 \ldots d_h^j\}, v_h}{\textbf{bvconcat}\ bv_l\ bv_h \Downarrow [\underbrace{\{d_l^0 \ldots d_l^k\}}_{\text{size of } bv_l\ \text{times}}\ \underbrace{\{d_h^0 \ldots d_h^j\}}_{\text{size of } bv_h\ \text{times}}], \texttt{concat}\ v_l\ v_h}\ \text{CONCAT TERM AND TERM}$$

Figure 4.15: Dependencies of byte-level preserving operations

$$\frac{bv \Downarrow d, v}{\textbf{bvneg}\ bv \Downarrow \texttt{squash}\ d, -v}\ \text{BVNEG}$$

$$\frac{bv \Downarrow d, v}{(\diamond_{bv} \in \textbf{rotate\_left} \mid \textbf{rotate\_right})\ n\ bv \Downarrow \texttt{squash}\ d, \diamond\ n\ v}\ \text{ROTATE}$$

$$\frac{bv_1 \Downarrow d_1, v_1 \qquad bv_2 \Downarrow d_2, v_2}{\textbf{bvcmp}\ bv_1\ bv_2 \Downarrow \texttt{squash}\ d_1\ d_2, \texttt{bvcmp}\ v_1\ v_2}\ \text{BVBNOP1}$$

$$\frac{bv_1 \Downarrow d_1, v_1 \qquad bv_2 \Downarrow d_2, v_2}{(\diamond_{bv} \in \textbf{bvadd} \mid \textbf{bvsub} \mid \textbf{bvmul} \mid \textbf{bvdiv} \mid \textbf{bvrem})\ bv_1\ bv_2 \Downarrow \texttt{squash}\ d_1\ d_2, v_1 \diamond v_2}\ \text{BVBNOP2}$$

$$\frac{bv_1 \Downarrow d_1, v_1 \qquad bv_2 \Downarrow d_2, v_2}{(\diamond_{bv} \in \textbf{bvshl} \mid \textbf{bvashr} \mid \textbf{bvlshr})\ bv_1\ bv_2 \Downarrow \texttt{squash}\ d_1\ d_2, v_1 \diamond v_2}\ \text{BVBNOP3}$$

$$\frac{bv_1 \Downarrow d_1, v_1 \qquad bv_2 \Downarrow d_2, v_2}{(\diamond_{bv} \in \textit{<bv\_cmp>})\ bv_1\ bv_2 \Downarrow \texttt{squash}\ d_1\ d_2, v_1 \diamond v_2}\ \text{BVCMP}$$

$$\frac{bl_c \Downarrow d_c, 1 \qquad bv_t \Downarrow d_t, v_t}{\textbf{if}\ bl_c\ \textbf{then}\ bv_t\ \textbf{else}\ bv_e \Downarrow \texttt{squash}\ d_c\ d_t, v_t}\ \text{BVITE THEN}$$

$$\frac{bl_c \Downarrow d_c, 0 \qquad bv_e \Downarrow d_e, v_e}{\textbf{if}\ bl_c\ \textbf{then}\ bv_t\ \textbf{else}\ bv_e \Downarrow \texttt{squash}\ d_c\ d_e, v_e}\ \text{BVITE ELSE}$$

$$\frac{bl_c \Downarrow d_c, \_ \qquad bv_t \Downarrow d_t, \_ \qquad bv_e \Downarrow d_e, \_}{\textbf{if}\ bl_c\ \textbf{then}\ bv_t\ \textbf{else}\ bv_e \Downarrow \texttt{squash}\ d_c\ d_t\ d_e, nondet + 0}\ \text{BVITE DEFAULT}$$

Figure 4.16: Dependencies of non byte-level preserving operations

Example

We illustrate the different analyses with the example from Figure 4.1. Figure 4.17 shows its actual trace, expressed with the formula language.

```
1  declare __memory_0;
2  declare __mem_ext;
3  // some setup which we removed from the trace for readibility
4  define __memory_8 = store 4 __memory_7 (bvsub __esp_8 0x4) 0x4; // sizeof(int) -> stack
5  define __esp_9 = bvsub __esp_8 0x4;
6  // &x -> stack
7  define __memory_9 = store 4 __memory_8 (bvsub __esp_9 0x4) (bvsub __ebp_1 0x3c);
8  define __esp_10 = bvsub __esp_9 0x4;
9  define __memory_10 = store 4 __memory_9 (bvsub __esp_10 0x4) 0x0; // 0 -> stack
10 define __esp_11 = bvsub __esp_10 0x4;
11 define __esp_12 = bvsub __esp_11 0x4;
12 // call read, store ret address
13 define __memory_11 = store 4 __memory_10 __esp12 0x8048830;
14 // read stub
15 define __buf_0 = select 4 __memory_11 (bvadd __esp12 0x8);
16 define __tmp_0 = select 4 __mem_ext 0;
17 define __memory_12 = store 4 __memory_11 __buf_0 __tmp_0;
18 define __eax_0 = 0x4;
19 define __esp_23 = (bvadd __esp_22 0x4);
20 // end stub
21 define __esp_24 = (bvadd __esp_23 0x10);
22 // same thing with buf = -0x38(ebp) for y, -0x34 for z, -0x30 for t and -0x2c for z
23 define __eax_1 = bvadd (select 4 __memory_32 (bvsub __ebp_1 0x3c)) 0x3; // a = x + 3
24 assert (bvsle __eax_1 0x4); // assert (a <= 4)
25 // b = y
26 define __eax_2 = select 4 __memory_32 (bvsub __ebp 0x38);
27 define __memory_33 = store 4 __memory_32 (bvsub __ebp_1 0x28) __eax_2;
28 // c = t
29 define __eax_3 = select 4 __memory_33 (bvsub __ebp 0x30);
30 define __memory_34 = store 4 __memory_33 (bvsub __ebp_1 0x24) __eax_3;
31 // assert (b != z)
32 assert (bvdiff
33           (select 4 __memory_34 (bvsub __ebp_1 0x28))
34           (select 4 __memory_34 (bvsub __ebp_1 0x34)));
35 // assert (c = v)
36 assert (bveq
37           (select 4 __memory_34 (bvsub __ebp_1 0x24))
38           (select 4 __memory_34 (bvsub __ebp_1 0x2c)));
39 define __memory_35 = store 4 __memory_34 (bvsub __ebp_1 0x28), 0x3; // b = 3
40 assert (bvsgt (select 4 __memory_35 (bvsub __ebp_1 0x30)) 0xa); // assert (t > 10)
41 assert (bvsle (select 4 __memory_35 (bvsub __ebp_1 0x2c)) 0x2d); // assert(v <= 45)
42 define __memory_36 = store 4 __memory_35 (bvsub __ebp_1 0x24) 0xa; // c = 10
```

Figure 4.17: Full trace from Figure 4.1

Let us start with the first condition: **assert** (bvsle __eax_1 0x4). Since the condition is BvCmp (BvSle, bv, BvCst(4)), analyzeCond calls analyzeValues(BvSle, bv, 4). The algorithm will apply inference rules to $\_\_eax\_1 \in [\![min; 4]\!]$, where $min = -2^{31}$. Following the reasoning from Figure 4.18, we conclude that the value of the first four bytes of the input is constrained by $[\![min; 1]\!]$.

$eax_1 \in [\![min; 4]\!]$

$\Leftrightarrow$ **bvadd (select** $4\ memory_{32}$ **(bvsub** $ebp_1\ 0x3c$**))** $3 \in [\![min; 4]\!]$ VAR on eax_1, l. 21

$\Leftrightarrow$ **select** $4\ memory_{32}$ **(bvsub** $ebp_1\ 0x3c$**)** $\in [\![min; 1]\!]$ ADD CST

$\Leftrightarrow$ $tmp_0 \in [\![min; 1]\!]$

SELECT, stored l. 18 ($buf_0 = ebp_1 - 0x3c$), and no condition for index

$\Leftrightarrow$ **select** $4\ mem\_ext\ 0 \in [\![min; 1]\!]$ VAR

$\Leftrightarrow$ $(0, 4) \in [\![min; 1]\!]$

INPUT, and no condition for index

Result: $mem\_ext[0..3] \in [\![min; 1]\!]$

Figure 4.18: Value analysis on $eax\_1 \in [\![min; 4]\!]$

The next condition is **assert** (bvdiff, select (...), select (...)) , which corresponds to $b \neq z$. Since it is an inequality, analyzeCond calls the dependency analysis. By following the reasoning from Figure 4.19, we conclude that the condition depends on bytes 4 to 7 and 8 to 11.

The following condition is **assert** (bveq, select (...), select (...)) , aka $c = v$ We cannot negate it because $c \neq v$ cannot be expressed with EECL, unless we already have a solution. If we were targeting it, on the other hand, we would use the equality analysis: analyzeEquality(Select(...), Select(...)). Following the reasoning from Figure 4.20, we would conclude that the bytes 12 to 15 are equal to the bytes 16 to 19.

Properties

Theorem 4.4.1 (Correctness)**.** *Path predicates expressed in EECL by using the analyses described previously are correct-enough.*

Proof. For each condition $cond$:

- When possible, we create the constraint using the equality or value analyses, which compute $[\![cond]\!]_{EECL}$ without losing any information. Since it is a direct translation, it is inherently correct: if the input variables satisfy the constraint, they will automatically satisfy the condition.

- When this is not possible, we either abandon the condition, ensuring we do not create an incorrect predicate, or we fall back on the dependency analysis. When relying on the dependency analysis, we constrain the input variable to its valuation in our current seed, which we know satisfies the condition.

$$\text{(4) VAR}\ \frac{var_{def}[tmp_1] = \dfrac{\text{CST}\ \dfrac{}{\texttt{0x4} \Downarrow [\{\}\{\}\{\}\{\}], 4} \qquad \texttt{merge}\ \{\}\ [\{4\}\{5\}\{6\}\{7\}] = [\{4\}\{5\}\{6\}\{7\}] \qquad input[4] = y = 4}{\textbf{select}\ 4\ mem\_ext\ 4 \Downarrow [\{4\}\{5\}\{6\}\{7\}], 4}\ \text{INPUT}}{tmp_1 \Downarrow [\{4\}\{5\}\{6\}\{7\}], 4}$$

$$\text{(3) VAR}\ \frac{var_{def}[eax_2] = \dfrac{\text{BVBNOP}\ \dfrac{\dots}{\textbf{bvsub}\ ebp_1\ \texttt{0x38} \Downarrow \{\}, nd_0 - 56} \qquad mem_{def}[mem_{32}, nd_0 - 56, 4] = \dfrac{\dots}{tmp_1 \Downarrow [\{4\}\{5\}\{6\}\{7\}], 4}\ \text{(4) VAR}}{\textbf{select}\ 4\ mem_{32}\ \textbf{(bvsub}\ ebp_1\ \texttt{0x38}\textbf{)} \Downarrow [\{4\}\{5\}\{6\}\{7\}], 4}\ \text{SELECT}}{eax_2 \Downarrow [\{4\}\{5\}\{6\}\{7\}], 4}$$

$$\text{(2) BVSUB}\ \frac{\text{NONDET}\ \dfrac{ebp_1 \notin var_{def}}{ebp_1 \Downarrow [\{\}\{\}\{\}\{\}], nd_0 + 0} \qquad \dfrac{}{\texttt{0x28} \Downarrow [\{\}\{\}\{\}\{\}], 40}\ \text{CST}}{\textbf{bvsub}\ ebp_1\ \texttt{0x28} \Downarrow \{\}, nd_0 - 40}$$

$$\text{(1) SELECT}\ \frac{\text{(2) BVSUB}\ \dfrac{\dots}{\textbf{bvsub}\ ebp_1\ \texttt{0x28} \Downarrow \{\}, nd_0 - 40} \qquad mem_{def}[mem_{34}, nd_0 - 40, 4] = \dfrac{\dots}{eax_2 \Downarrow [\{4\}\{5\}\{6\}\{7\}], 4}\ \text{(3) VAR}}{sel_1 \Downarrow [\{4\}\{5\}\{6\}\{7\}], 4}$$

$$\text{BVCMP}\ \frac{\text{(1) SELECT}\ \dfrac{\dots}{sel_1 \Downarrow [\{4\}\{5\}\{6\}\{7\}], 4} \qquad \dfrac{}{sel_2 \Downarrow [\{8\}\{9\}\{10\}\{11\}], 5}\ \text{SELECT}}{\textbf{bvdiff (}\underbrace{\textbf{select}\ 4\ mem_{34}\ \textbf{(bvsub}\ ebp_1\ \texttt{0x28}\textbf{))}}_{sel_1}\ \underbrace{\textbf{(select}\ 4\ mem_{34}\ \textbf{(bvsub}\ ebp_1\ \texttt{0x34}\textbf{))}}_{sel_2} \Downarrow \{4; 5; 6; 7; 8; 9; 10; 11\}, 0}$$

Result: depends on bytes 4 to 11, $mem\_ext[4..7] = 4 \wedge mem\_ext[8..11] = 5$.

Figure 4.19: Dependency analysis on **select** $\neq$ **select**

select $4\ memory_{34}$ **(bvsub** $ebp_1\ 0x24$**)** = **select** $4\ memory_{34}$ **(bvsub** $ebp_1\ 0x2c$**)**

	alias(**select** $4\ memory_{34}$ **(bvsub** $ebp_1\ 0x24$**)**)	
=	alias($mem_{def}[4, memory_{34}, ebp_1 - 36]$)	select
=	alias(eax_3)	stored line 28
=	alias(**select** $4\ memory_{33}$ **(bvsub** $ebp_1\ 0x30$**)**)	defined line 27
=	alias(tmp_3)	stored on 4th read
=	alias(**select** $4\ mem\_ext\ 12$)	defined on 4th read
=	Some$(12, 4)$	user input
	alias(**select** $4\ memory_{34}$ **(bvsub** $ebp_1\ 0x2c$**)**)	
=	alias(tmp_4)	defined on 5th read
=	alias(**select** $4\ mem\_ext\ 16$)	defined on 5th read
=	Some$(16, 4)$	user input

Result: $mem\_ext[12..15] = mem\_ext[16..19]$

Figure 4.20: Equality analysis on **select** = **select**

This ensures that any solution to $[\![cond]\!]_{EECL}$ will satisfy $cond$. We create the path predicate by doing the conjunction of the correct constraints obtained with either of the analyses. As this only reduces the search space of the constraint, we will not lose any information. The only information we lose are difference constraints, which removes only one point from the solution space. The resulting path predicate will be *correct*. □

Implementation

In this section we will first discuss what we implemented as part of the BINSEC tool, before presenting two implementation details specific to the LSE, which allows us to keep the constraint inference as efficient as possible.

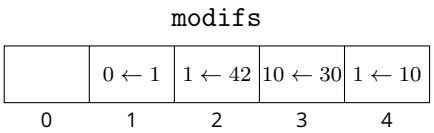
Difference with Theory Even if we defined and formalized 3 different analyses, in practice we only implemented two: considering the added engineering effort required, the equality analysis remains theoretical for now. But we do believe that adding it would improve LSE even further, and consider it one of the priorities of future work.

Memory Representation

The memory in the trace is a table, and each modification creates a new version. As such, we can represent the memory as the ordered list of modifications. Finding the element at an index is then a matter of following the list from the last modification to the first, stopping once we find when the value at the index was set.

```
declare mem0;
define mem1 = store 1 mem0 0 1;
define mem2 = store 1 mem1 1 42;
define mem3 = store 1 mem2 10 30;
define mem4 = store 1 mem3 1 10;
```

Sample trace



List representation

0	$\{1 \mapsto 1\}$
1	$\{2 \mapsto 42; 4 \to 10\}$
10	$\{3 \mapsto 30\}$

Map representation

Figure 4.21: Example of memory representations

We illustrate the possible memory representations using the code from Figure 4.21a. Figure 4.21b shows the list representation we are currently discussing. Let us imagine we are looking for the element stored at the index 10, in mem_4 – aka **select** $1\ mem_4\ 10$. Given the list `modifs`, we would start at `modifs[4]`, which is the last modification for this memory state. Going through the list backward, we would stop at `modifs[3]`, when 30 was stored at the index 10.

The drawback of this technique is that it requires systematically backtracking on the list of store instructions anytime we encounter a select, which happens often – think of push and pop instructions. To simplify the analysis, we decided to add a pre-processing step in order to build a more straightforward representation of the memory. In this representation, instead of associating the modifications to the memory state number, we sort them by the access index first. In practice, we compute the value (be it symbolic or concrete) of each index, and build a hashtable associating each index to a map, which itself associates the memory states to the element stored. In the case of the example, this gives us the representation from Figure 4.21c. To resolve **select** $1\ mem_4\ 10$, we now get the modifications at index 10 (after evaluating the bvterm, which is a constant here): $\{3 \mapsto 30\}$. This technique is inherently more efficient than the first one, as we only look through modifications to the interesting index, rather than all of them.

Table 4.1: Example of split term in memory

4	$\{i \mapsto eax\{0..7\}\}$
5	$\{i \mapsto eax\{8..15\}\}$
6	$\{i \mapsto eax\{16..23\}\}$
7	$\{i \mapsto eax\{24..31\}\}$

NB: in order to simplify the lookup into the map, we split multi-bytes selects and stores. As such, **store** $4\ mem_i\ eax$ will be added to the map as described in table 4.1. And when doing **select** $4\ mem\ 4$ we will just individually lookup bytes 4 to 7 from memory, and concatenate them.

Caching information

When applying the dependency analysis, we can reuse dependency and value information. We do this by creating a cache where we store variables' dependencies and values once we have computed then once. When analyzing a variable, we will first check whether it is in the cache, and only backtrack if it is not. This allows us to never backtrack more than once on each variable, thus making the complexity of the dependency linear wrt the size of the trace. However, this does not apply to the value analysis since the analysis depends on the constraint. In fact, $eax \in [\![4; 4]\!]$ and $eax \in [\![5; 15]\!]$ would not have the same result, so we cannot reuse the dependency for the first one in order to compute the second one.

Discussion

LSE usage

The constraints generated by LSE can be used in different ways:

on their own by using an enumeration algorithm, as described in enumTests, LSE is stand-alone. We use inverting mode to explore new paths, when possible, then target each new path, enumerating solutions to explore it. However enumeration, while exhaustive, is not necessarily the most efficient way to explore a program's space.

combined to fuzzing this is the technique we implemented in our tool CONFUZZ, described in Chapter 5. When targeting a constraint (hence a new path), instead of enumerating solutions, we use a constrained fuzzer: a modified greybox fuzzer which only generates test cases that satisfy a

given constraint. This allows us to reuse fuzzing's efficient test generation techniques, while constraining them to a specific part of the program.

Constraint Language

In theory, our constraint language deals with two types of constraints: equality and integer interval. This lets us keep the resolution simple, but here we will discuss possible extensions.

- $x \neq c$: to add difference between the input and an integer, we would need to consider sets of intervals: $x \neq 4$ would translate to $x \in \{[\![min; 3]\!]; [\![5; max]\!]\}$. The constraint language would then be $\bigwedge_i \ x_i \in \widehat{I_i} \wedge \bigwedge_{i,j} x_i = x_k$ where $\widehat{I_i} = \{I_i^0, \ldots, I_i^n\}$ is a set of disjoint intervals, with its normalized version being $\bigwedge_{l \in L} l \in \widehat{I_l} \quad \wedge \ \bigwedge_{x \in X \setminus L} x = l_x$. This is still fast-enumerable: when enumerating solutions, instead of simply incrementing the input until it reaches the greater bound of the interval, we would go through the list of possible values. However, normalizing the formula would be more complicated. In particular, when merging the constraints from equal variables, we would need to go through each variable's set of interval constraints to compute the intersection. Given the fact that difference conditions only remove one element from the solution space, we have decided not to treat them, and to skip them instead.

- $x \neq y$: could be added to the constraint language, and for it to still be easily-enumerable. This is due to the fact that we set leaders one at a time: once x has been set to v, the constraint becomes $y \neq v$, where v is a constant. We then only need to remove v from the possible values of y, until we next backtrack to x.

Limitations and perspectives

Our technique is limited by the concretization we use when unable to create a more accurate EECL constraint. In this case, we look up which input the condition depends on, and constrain those to their current value which we know satisfy the condition – at least for targeted constraints. While this technique ensures that the constraint will be "correct-enough" (correct except for difference constraints), it also drastically reduces the solution space for said variables. However, we argue that since this is only our fallback, we still explore more than standard concrete symbolic execution would. Furthermore, since we continue targeting previous conditions, we will end up trying different values for the constrained input, and hopefully explore more paths which satisfy this condition.

In the future, a natural extension would be to add equality between variables. This would require implementing the equality analysis using the same kind of pattern matching as the other analyses. We might then want to look further into augmenting the constraint language's expressivity. We could for example think of adding more complex relations between parts of the input. To do this we would need to first formally define a new constraint language, prove it is correct by modifying the enumeration algorithm, then decide how to infer the new types of constraints.

Related Work

Symbolic Execution depends on two steps: creating the path predicate, and solving it to generate a solution. As such, variants of symbolic execution usually aim to increase efficiency in one of those two key areas.

Constraint Solving Among improvements to constraint solving, we can quote constraint elimination [58] – removing either irrelevant sub-constraints when adding the latest one, or duplicates – and incremental solving [58, 8] – storing in a cache the result of path requests, so as to reuse them when possible. However, those techniques are not relevant to Lightweight Symbolic Execution: as we create easily-enumerable constraints *by design*, we do not require an additional simplification step.

Path Predicates can be made simpler either by under-approximating or over-approximating them, using abstraction or concretization techniques [23].

Over-approximating happens when the path predicate is under-constrained. This results in a complete but not correct predicate: it accepts solutions that do not follow the path. EXE [10] uses such a technique, but only as a temporary way to simplify the constraint. When trying to solve a constraint with memory access, which are hard to solve because they involve array theory, it first removes all array access from the constraint. This creates an over-approximation, which is then sent to the regular solver. If the answer is UNSAT, it means that the original constraint cannot be satisfied either. Otherwise, the solution given by the solver might not be satisfy the original constraint. EXE then adds back the array access to the constraint, one-by-one, until it gets an UNSAT result or an answer for the original constraint. As such, while it does use over-approximation, it is only used to weed out unsatisfiable constraints early on, and in the end any solution is computed using the precise path predicate. When it comes to over-approximations, we can also mention Pangolin [39]. Pangolin combines a symbolic analysis with fuzzing, but works on abstractions of the path predicates, for

which solutions are created using an out-of-the-box sampling technique. This abstraction is an over-approximation of the path predicate, as they prioritize having all test cases that follow the path be a solution to the path predicate, whereas ConFuzz ensures that only test cases that follow the path are solutions to the path predicate. This is one of the key – and novel – elements of ConFuzz, because it means we can efficiently drive the fuzzing past hard-to-solve conditions, where Pangolin would keep on creating unsatisfactory test cases.

Under-approximations is introduced by concretization, when inputs of the program are set to a single concrete value, rather than kept symbolic. This is a technique specific to Concolic Testing, where the program is executed on a test case as well as symbolically. DART [35] is one of the first tools to have thus used dynamic analysis on a concrete test case to help symbolic execution. It builds test cases incrementally, starting with a random concrete input. The program is then executed both concretely and symbolically on the test case, meaning that DART follows the path of the concrete execution, but collects information about the symbolic variables and expressions at each instruction. Each time it encounters a condition in the path, it checks in a cache whether the branch corresponding to the negation of the condition was explored, and if not it tries creating a test case that does. Furthermore, whenever DART finds itself unable to reason about the symbolic expressions, be it because library code is unavailable or the solver cannot solve the constraint, it replaces the expression by its concrete value. CUTE [58] took this idea one step further, introducing a logical input map to represent inputs more precisely. This allows the analysis to more effectively track symbolic input, including pointer values. As a result, their method can be used on a wider range of programs, by solving constraints that are unfeasible for DART. While ConFuzz also sets the input to a concrete value when unable to reason more precisely about the condition, this is only done when targeting an interesting transition, allowing us to create multiple test cases which follow this transition. As for library calls, they are stubbed in order to avoid resorting to concretization, except for special cases where it does not matter, such as memory allocations.

Conclusion

In this section, we presented Lightweight Symbolic Execution. An alternative to Concrete Symbolic Execution, it analyzes execution traces of the PUT, computing easily-enumerable path predicates. Those path predicates lead to specific conditions on the path, either targeting new, interesting branches, or negating conditions to try reaching brand new parts of the code. Easily-enumerable constraints are a concept we define: the complexity of enumerating solutions is

linear wrt the size of the input. In practice, we use a specific constraint language, which we prove to be easy-enumerable, as well as explaining how we infer constraints from the trace's conditions. Finally, we discussed the limitations and perspectives of our technique, before comparing it to other symbolic execution techniques.

Chapter 5

ConFuzz, Combining Lightweight Symbolic Execution with Constrained Fuzzing

Contents

In this chapter, we describe how we combined Lightweight Symbolic Execution (LSE) with a tailor-made *Constrained Fuzzer*, a modification of the AFL greybox fuzzer which creates test cases according to a given predicate. We first present how both tools work together (Section 5.1). We then describe the constrained fuzzer itself (Section 5.2) as well as the key components of the combination: how the information about the trace (Section 5.3) and the predicates is exchanged (Section 5.4). Finally, we present our experimental evaluation on a standard benchmark from the literature (Section 5.5). In each section, we will give details about the implementation, and we will discuss limitations (Section 5.6) and related work (Section 5.7) at the end.

Overview

Lightweight Symbolic Execution, as described in Chapter 4, takes an execution trace of the Program Under Test (PUT), as well as a target branch, and returns a predicate that targets the branch, as well as possibly several predicates which invert the following branching conditions.

On the other side, our constrained fuzzer receives predicates, creates test cases which satisfy the constraints from the predicates, runs the program on those and determines whether a new branch was taken. If so, the trace and the branch are sent to the LSE. Otherwise, it keeps going until it creates a test case that does, or we stop it.

This communication loop is described in Figure 5.1. In practice, the communication is asynchronous, as both techniques run in parallel. The fuzzing, in particular, runs continuously. Similarly to AFL, the fuzzer keeps a database of interesting test cases. Target predicates are added to the relevant test case in the database, while inversion predicates are immediately solved. On the other hand, the LSE is in sleep mode until it receives a trace, which is then immediately analyzed. Both types of information are stored until the receiver is available, ensuring we never lose messages.

How To Create Solutions

Here we first give insight into how AFL creates new test cases, before explaining how we modified it to create constrained test cases.

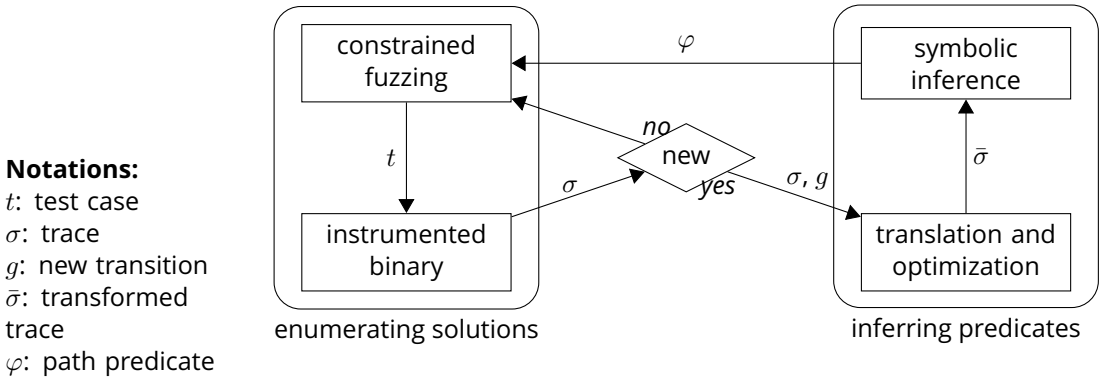


Figure 5.1: Overview of ConFuzz

AFL

In order to create a new test case, AFL picks one from its database, and mutates it. The selection is based on different information about the test cases, such as whether they were found recently, or if they already led to the discovery of new code. If it is the first time this test case is being fuzzed, AFL first goes through the deterministic phase. During this phase it systematically goes through every byte of the test case and applies the mutations, one at a time. AFL then goes into havoc mode. This is described in algorithm 1.

The mutation engine mutates a random number of bytes, choosing a random mutation for each offset (the list is in appendix A). The PUT is then ran on the test case created by the multiple mutations, to check whether it reaches a new part of the code. If not, the mutation engine will reset the test case, before mutating it again. This loop also happens a random number of times, before the engine switches to the next test case.

ConFuzz

Targeting mode When we have a target predicate, we want to create multiple test cases which satisfy it. To do this, we leverage AFL's fuzzing mutation engine. We keep the test case selection mechanism, since we also prioritize coverage, and only modified the deterministic phase to skip over offsets with dependency constraints. Our goal is for test cases generated by the havoc phase to satisfy the constraints. We achieve this by modifying the havoc algorithm as described in Algorithm 2.

We consider three types of constraints, all of which correspond to $x \in [\![a; b]\!]$ from Definition 4.2.2:

dependency constraint when the constraint comes from dependency, it means

Algorithm 1: AFL - havoc stage

```
   Data: buffer in_buf of len len containing the original test case
 1 memcpy(in_buf, out_buf, len);
 2 stage_max ← f(perf_score, ..);
 3 if stage_max < HAVOC_MIN then stage_max ← HAVOC_MIN;
 4 temp_len ← len;
 5 for stage_cur = 0 to stage_max − 1 do
 6     use_stacking ← rand(2, 4, 8, 16, 32, 64, 128);
 7     for i = 0 to use_stacking − 1 do
 8         pos_modif ← choose_havoc_pos(temp_len);
           // applies a random mutation to pos_modif, cf algo 4,
              appendix A
       // stop there if we found an interesting entry
 9     if common_fuzz_stuff(out_buf, ...) then goto abandon_entry;
       // restore out_buf
10     if temp_len < len then out_buf ← realloc(out_buf, len);
11     temp_len ← len;
12     memcpy(out_buf, in_buf, len);
```

we are constraining the input to its current value. To satisfy these constraints, we only need to not modify those offsets. This is done line 8, where we simply skip dependency constrained offsets when choosing which offset to mutate, using the function from Figure 5.2.

equality constraint when mutating an offset to which an equality constraint applies, we set the offset to the constrained value and keep going (lines 9-11). We do this because there is no use trying to mutate it, when there is a single valid value.

interval constraint we let AFL mutate the offset as usual, so as to use the smart mutations. Then, as seen on lines 12-14, we check the value at the offset after the mutation, and if the result is now outside of the interval, we bring it back by setting it to a random value of the interval.

Inverting mode In this case, the seed does not satisfy the path predicate, since we are trying to reach a new part of the code by taking a new transition. Our goal is to create a single seed that does, and run the program on it to check if it leads to a new part of the program. We create said seed by mutating the interesting seed t_0, taking advantage of the fact that it reached the condition,

Algorithm 2: ConFuzz - creating multiple solutions to target predicate

Data: buffer in_buf of len len containing the original test case, dep_cstr the set of dependency constraints, val_cstr the set of value constraints

```
1  memcpy(in_buf, out_buf, len);
2  stage_max ← f(perf_score, ..);
3  if stage_max < HAVOC_MIN then stage_max ← HAVOC_MIN;
4  temp_len ← len;
5  for stage_cur = 0 to stage_max − 1 do
6      use_stacking ← rand(2, 4, 8, 16, 32, 64, 128);
7      for i = 0 to use_stacking − 1 do
8          pos_modif ← choose_havoc_pos(temp_len, dep_cstr);
9          if val_cstr[pos_modif] exists and is ⟦v; v⟧ then
               // we have an equality constraint
10             out_buf(pos_modif) ← v;
11             continue;
           // applies a random mutation to pos_modif, cf algo 4,
              appendix A
12         if val_cstr[pos_modif] = val_cstr exists then
13             if out_buf[pos_modif] ∉ val_cstr then
14                 out_buf[pos_modif] ← rand(val_cstr);
       // stop there if we found an interesting entry
15     if common_fuzz_stuff(out_buf, ...) then goto abandon_entry;
       // restore out_buf
16     if temp_len < len then out_buf ← realloc(out_buf, len);
17     temp_len ← len;
18     memcpy(out_buf, in_buf, len);
```

```
static u32 choose_havoc_pos(u32 limit, u8 target) {

  u32 nb_dep = queue_cur->nb_dep;
  u32* dep_cstrs = queue_cur->dep_cstrs;

  // if no dependencies, nothing to do
  if (nb_dep == 0) return UR(limit);

  // count the number of forbidden values < limit
  u32 n = 0;
  while (n < nb_dep && dep_cstrs[n] < limit) n ++;

  // if no possible value, stop
  if (n >= limit) return -1;

  // get random value, < nb of values to choose from
  u32 u = UR(limit-n);

  // increment u for each forbidden dependency smaller than it
  u32 res = u;
  u32 ind = 0;
  while (ind < n && dep_cstrs[ind] <= res) {
    res++;
    ind++;
  }

  return res;
}
```

Figure 5.2: Code for choose_havoc_pos

even if it did not satisfy it. We then modify the constrained offsets to values inside the intervals. This process is described by algorithm 3.

Algorithm 3: ConFuzz - solving inversion predicate

Data: buffer in_buf of len len containing the original test case, val_cstr the set of value constraints

1 out_buf $\leftarrow$ in_buf;
2 **foreach** $(pos, inter) \in$ val_cstr **do** out_buf[*pos*] $\leftarrow$ `mid`$(inter)$;
3 `common_fuzz_stuff`(out_buf, ...);

Implementation

We directly modify AFL's code, which is written in C and originally 7.5kloc, highly optimized and barely commented. This requires to first reverse-engineer the algorithm used by AFL, by looking at the code, before modifying it where appropriate. This lead to adding around 4kloc.

The Trace

There are two steps to retrieving the trace: first we need to get the trace at runtime, then we need to pre-process the trace to translate it into the formula language from REF.

Retrieving the Trace

AFL considers the code to be divided into basic blocks: sequences of code which do not contain branches, except for entry and exit. In this configuration, we can represent the trace as a list of transitions between basic blocks. This is enough to precisely describe the execution path, since we know exactly which parts of the code were taken. AFL retrieves this information by instrumenting the PUT at compile-time. The instrumentation adds trampolines at each jump and possible jump target location. Those delimit basic block, and at each trampoline, the transition is logged into shared memory. Once the execution is finished, the shared memory contains the list of transitions that were taken. The fuzzing engine can then access this information, and uses it to determine whether any new transition was taken. However, we cannot directly use this information, because the transition from block A to block B is logged as a hash: $id(A->B) = (id(B) >> 1)$ xor $id(A)$, while we need to know exactly which basic blocks were taken.

Instead, we extended the shared memory, so as to store our own list of basic blocks ids next to AFL's trace. We also added new trampolines to the instrumentation, for example at calls to read to retrieve the size of the read, or at calls to allocation functions to get the address of the allocated memory. Once the execution has ended, we let AFL determine whether the test case is interesting. If yes, we get the hash of the transition from AFL's analysis, and the trace from the shared memory, and send them both to the LSE.

Transforming the trace

In order to analyze the trace as described in REF, we must create it from the list of basic block ids. The first step, done once, when starting ConFuzz – rather than for each trace – is to disassemble the PUT's binary file. We do this using Binsec's disassembly, which gives us a DBA [26] representation of all of the file's code. At this point, we still need to determine what the basic blocks are, to match them to the list. We do this using labels added by the instrumentation along with the trampolines, which delimit each basic block and gives us the id. This information is preserved by the compilation, and allows us to divide the DBA code into the original basic blocks. Once we have the id to DBA code sequence association table, building the DBA trace is a matter of matching basic blocks' ids to the corresponding DBA code block. We simply do this by going through the list of ids. At this point, we also go through the code of each basic block, in order to clean them up. For example, we use the next block's id to determine the result of conditional jumps, in order to replace them with assertions. We also identify calls to external functions – calls to functions whose code we do not have, and determine which type of stub to use.

This gives us the trace as a list of DBA instructions, and we also use the blocks ids to identify the interesting branch. We then call a translation function, which goes through the list of DBA instructions and builds the equivalent trace in the formula language described in Section 4.4. We then call the optimization functions added to Binsec by Farinier [30], which highly simplify the formula.

At this point, we have a trace which we can analyze, and we also know which assertion corresponds to the result of the target branch. We communicate those to the LSE, along with the concrete values from the execution, as shown in Figure B.1.

Implementation

We modified AFL's instrumentation functions in order to add our own information. We did this by slightly modifying the assembly code added to the compiled

file by afl-gcc, in order to call our own functions, which we wrote in C and link to the file at compile-time.

The transformation functions, on the other hand, are part of the Binsec framework. We re-use Binsec's disassembly, as well as the syntax definition for DBA and formula, and formula's optimization function. However, the transformation functions themselves were written from scratch in OCaml – 1kloc.

Finally, two options affect the trace:

entry point defines where the trace starts. By default, this is the start of the main function, but we might want to change it to bypass initialization functions.

shadowed blocks let us indicate parts of the code which will not appear in the trace. Those obviously depend on the code, but this allows us to ignore irrelevant parts of the code.

Both those options allow us to leave sections of the code out, to keep the trace's size's small.

In particular, blocks that are shadowed are ignored when building the trace at runtime. This is done by communicating the list of shadowed regions – defined by the id of the first and last block – to the code added by the instrumentation through a dedicated shared memory. Said code will then keep track of whether it is in a shadowed regions, using a stack to deal with nested regions. It is essential to ignore those blocks as early as during runtime, because the amount of space in which we store the trace is limited, and we want to avoid losing the last basic blocks.

Communicating the Predicates

Predicate format

As a reminder, in practice the constraint language is made of value constraints and dependency constraints, which make up either target predicates or inversion predicates. Both types of predicates are represented the same way and differ by a boolean, as shown in Figure B.2.

Dependency constraints are represented as a list of offsets that should not be modified. Value constraints are represented by a record, defining both the interval of constrained offsets, and the interval of values they are constrained to.

We communicate this information to the constrained fuzzer, along with the file name and the inversion boolean. As you can notice in Figure B.2, we differ-

entiate here between signed constraints and unsigned constraints, as the integers defining the value interval will be interpreted differently by the constrained fuzzer.

Reception of predicates

The fuzzer runs continuously, but same as AFL, it functions through an infinite loop of fuzzing cycles, during which a test case is fuzzed as described in algorithm 1. At the start of each cycle, our constrained fuzzer checks if the buffer contains predicates, and if so treats them immediately. The constrained fuzzer's behavior differs depending on the type of predicate it receives:

target predicates are added to the database. We do not necessarily want to solve them immediately – though we do immediately run a single fuzzing campaign – but rather we want to have access to the information whenever we fuzz the test case in the future.

inversion predicates are solved immediately, as we only need a single solution. We just want to see if the solution does take a new branch, which we let AFL determine. If it does, the solution test case is considered to be interesting, and treated as such.

Experimental Evaluation

Goals of the evaluation We investigate the following Research Questions.

RQ1 How does ConFuzz compare against standard fuzzing and symbolic execution tools on a standard benchmark?

RQ2 how does ConFuzz compare to existing tools that mix symbolic execution with fuzzing, on a standard benchmark?

Experimental Setup

We ran ConFuzz against the version of AFL [69] it was built on, as well as the state of the art in terms of fuzzing and symbolic execution – AFL++ [32] and KLEE [8] – over the standard fuzzing benchmark LAVA-M [28]. We also consider QSYM [68], which combines fuzzing and symbolic execution, as well as a combination of AFL++ and KLEE's results in order to approximate the results of a black-box combination of both tools (in the vein of Driller [59], not available on x86). Note that Pangolin [39] is not available.

LAVA-M is a set of four real-world programs, extracted from the GNU Coreutils, in which a number of faults (bugs) were automatically injected. It is commonly used for evaluating and comparing fuzzers. Due to an implementation problem in ConFuzz – namely we do not have a stub for who's particular read function – we consider only 3 programs out of 4. The main metric for comparison is the number of detected faults. For each tool and each program, we ran the seed generation process five times with a one hour time limit. Having several runs allows us to mitigate the effects of randomness. We report the average, minimal and maximal values, as well as the standard deviation and the *Vargha-Delaney statistic* ($\hat{A}_{12}$) [61]. This last metric has been used in several recent fuzzing papers [4, 13, 54], and measures the probability for a technique 1 (here, ConFuzz) to yield better result than a technique 2 (here, the other tools). Results are depicted in table 5.1.

ConFuzz, SE, Greybox Fuzzing

Table 5.1 shows the results for ConFuzz and the tools we compare ourselves to.

Table 5.1: Vulnerabilities detected in programs from LAVA-M (TO=1h, 5 runs)

# injected faults			AFL	AFL++	KLEE	A/K*	QSYM	ConFuzz
base64 - 3kloc	44	**Avg**	0	0.2	10.0	10.2	47.8	38.8
		Min	0	0	8	8	47	38
		Max	0	1	11	12	48	39
		Dev (σ)	0	0.4	1.3	1.5	0.4	0.4
		$\hat{A}_{12}$	1.0	1.0	1.0	1.0	0.0	-
md5sum - 3kloc	57	**Avg**	0	0	0	0	0	9
		Min	0	0	0	0	0	7
		Max	0	0	0	0	0	11
		Dev (σ)	0	0	0	0	0	1.7
		$\hat{A}_{12}$	1.0	1.0	1.0	1.0	1.0	-
uniq - 3kloc	28	**Avg**	0	0.4	5	5.4	17.4	26.9
		Min	0	0	5	5	12	15
		Max	0	1	5	6	25	29
		Dev (σ)	0	0.5	0	0.5	4.5	3.6
		$\hat{A}_{12}$	1.0	1.0	1.0	1.0	0.94	-

*: A/K denotes the combination of AFL++ and KLEE

RQ1: vulnerability detection Compared to standard fuzzing and symbolic execution, ConFuzz offers a clear improvement. Most notably, the fuzzers are barely able to find crashes, AFL++ finding at most one, due to the complexity of the bug conditions in LAVA-M. KLEE struggles as well, and finds less than a third of the vulnerabilities ConFuzz does.

RQ2: Comparison with other mixed tools While the number of vulnerabilities found when combining the results of AFL++ and KLEE is higher than that found by each technique on its own, it stays much lower than ConFuzz. This confirms that combining out-of-the-box fuzzers and symbolic executors, as Driller [59] does, does not yield enough improvement. QSYM has better results, as it outperforms ConFuzz with probability 1 on base64. Still, on md5sum it does not find any bugs and while its results on uniq are above average, ConFuzz clearly outperforms it. Interestingly, in one run ConFuzz finds 29 bugs, which is one more than expected by the LAVA-M developers.

Discussion

Our implementation choices introduce several technical difficulties, which for the most part we only lacked time to solve. We present them in this section, as well as other improvements we thought of for the fuzzing part of ConFuzz.

Trace Length

Given the fact that we need to go through the trace several times, both to transform it and to analyze it, we need to keep it from being too long. In particular, one of the examples gave us at first a trace of millions of block ids, which ended up too long to work with.

One way we work around this limitation is by limiting the size of the trace. First off, we cut the end of the trace: past the interesting transition, we only keep a certain as many blocks as branches we want to invert, since we know the last blocks are not relevant to the analysis. We also ignore the code executed before hitting the entry point, which we can define to be the start of any basic block. Finally, if we notice that a given block is repeated a lot – for example in a loop – and we determine that it does not have any side-effects, we can shadow it, and it will be skipped by the id to DBA function.

The optimization functions also help a lot with this issue, since they simplify the trace. They mostly do so by removing unused instructions (such as setting flags that are not used), propagating constants to simplify operations, and using read-over-write optimization on the memory to resolve select operations, wherever possible.

Communication

The current version of ConFuzz communicates via files: we use the fact that AFL has an out directory, and add two sub-directories, one each for traces and

predicates. When either of the tools has something to send to the other one, it creates a file, in which it writes the information – in binary format, to save space. Those sub-directories work as the buffer space for our asynchronous communication: the files are stored there until the tool reads them, at which point they are moved to another sub-directory so as to not be read twice. We could however improve this by using a shared memory instead of files: this would spare us the cost of writing and reading files. We could for example use circular buffers: two pointers would indicate the beginning and end of the information (which could be several traces or predicates). We would read from the first one and write to the last one, looping back to the beginning to reuse the space if it is available. When the buffer is full, we would just allocate a new, bigger one.

Constrained Fuzzing

We currently use constraints solely to create solutions. However, we could also use this information to guide the solution creation.

When an offset is constrained to an interval of value, we might want to try all of the values, for example during the deterministic phase. If we had $x \in [\![0; 5]\!]$ because there is a switch on x, we would immediately try all possibilities, without waiting for fuzzing to do so. Furthermore, in a bug-funding objective rather than coverage-guided, we might want to add a mutation which sets the offset to either of the bounds of the interval, or maybe $min - 1$ or $max + 1$, in order to test what happens there.

More generally, we could use the information that offsets are constrained as an indication that they are relevant to the path and thus should be fuzzed more. In practice, for target predicates, we would modify `choose_havoc_pos` so as to target the offsets with value constraints (since dependency constraints are not to be modified). As for inversion predicates, we could imagine falling back on the dependency analysis when we do not have a value constraint. Those offsets should be the primary targets of fuzzing, since we know they need to change to satisfy the inverted condition, even if we do not know how. So we could try creating a solution by setting the offsets with constrained values then running the havoc phase of fuzzing on the dependency constrained offsets, before discarding the predicate if the campaign does not succeed.

Related Work

Ever since fuzzing was invented, and even more since it was popularized by AFL [69], a multitude of tools have proposed techniques to improve it. We can

differentiate those based on whether they improve fuzzing as is, at most combining it with a static analysis, or combine fuzzing with symbolic execution, or modified and augmented fuzzing's expressivity using dynamic analysis.

Improved fuzzing Several research works have improved internal components of Greybox-Fuzzing. AFLFast [5] favors test cases covering rarely taken paths of the PUT, then introduces a power schedule to determine the time required to fuzz selected test cases. FairFuzz [46] introduces a mutation masking technique and changes test case selection strategy to increase code coverage. CollAFL [33] modifies the hash algorithm used to determine coverage, in order to get more precise information and increase the accuracy of the coverage. Steelix [48] uses static analysis and a modified instrumentation to find magic bytes and mutate test cases according to comparisons present in the program. Even though those techniques show encouraging results, they do not offer formal guarantees about the analyses used.

Hybrid Fuzzing Since the seminal work on Driller, many attempts have been directed toward combining fuzzing and symbolic execution. Yet, they perform mostly a shallow combination, with out-of-the-box tools, rather than a deep integration as we have explored.

Driller [59] alternates between both techniques, calling SE when fuzzing is "stuck" to help find seeds satisfying new conditions. Contrarily to our approach, they use a symbolic engine out-of-the-box, and only communicate by exchanging seeds. As a result, they lose the information gained from the path predicate, unlike ConFuzz who communicates approximated path predicates.

QSYM [68] runs a symbolic engine parallel to two instances of AFL. Its goal is to optimize the combination by relying on fuzzing to optimize the symbolic execution. In particular, the SE used by QSYM resorts to optimistic solving, i.e. it only keeps the last condition of the trace, arguing that the fuzzer will discard seeds that do not satisfy the rest of the path predicate. As a result, they sacrifice correctness, which we argue is important, especially when trying to guide the fuzzer towards a newly explored part of the code.

Pangolin [39] relies on QSYM to retrieve path predicates, builds a polyhedral abstraction of the predicate and then leverages an out-of-the-box sampling technique to generate seeds satisfying the approximated constraints. The main difference with our approach is that Pangolin aims at over-approximating path predicate computation, where our approach aims at under-approximations, in order to drive fuzzing towards hard-to-reach parts of the program. Actually, over-approximations has been well studied for a long time in software analysis, while designing under-approximations suitable to symbolic exploration and fast

solution enumeration is novel to this work.

Augmented Fuzzing Other techniques try to help fuzzing solve specific conditions without relying on symbolic execution, but without any formal guarantee. VUzzer [56] employs dynamic taint analysis (DTA) to extract control and data flow features from the PUT in order to guide the input generation. Angora [14] aims to satisfy a condition on a path by first identifying relevant input bytes through taint tracking, then finding a solution through gradient descent with the targeted condition as sole objective. Matryoshka [15] takes this one step further, using dependency flow analysis to identify all relevant conditions leading to the target condition, and taint analysis to identify the bytes that flow into the condition. Finally, Eclipser [16] relies on dynamic analysis to infer possible path predicate abstractions and then try to solve them through ad hoc patterns.

Conclusion

In this section, we explained how we combined the Lightweight Symbolic Execution to constrained fuzzing to create ConFuzz. Constrained fuzzing is a novel way to consider fuzzing, where instead of solely prioritizing coverage, the mutation engine fuzzes solutions to a predicate. This allows us to apply the efficiency of fuzzing to constraint solving, replacing the SMT solver from traditional Symbolic Execution.

We described how the final tool functions, and in particular how the constrained fuzzing and the predicate inference communicate. We also showed some of the engineering effort that went into creating ConFuzz, a tool which combines 4kloc of C and 6kloc of OCaml code.

We then gave an overview of the state of the art of fuzzing techniques, and how our tool compares to some of these tools. We also discussed the limitations of our technique, as well as many perspectives for future work.

Chapter 6

Conclusion and Perspectives

Conclusion

The work of this thesis focuses on automatically testing programs. The goal is to efficiently find crashes, thus bugs, by creating test suites that cover as many of the program's paths as possible. We propose to combine two of the more popular techniques, fuzzing and symbolic execution, by modifying both to deeply integrate them with each other. To achieve this, we create *Lightweight Symbolic Execution* and *Constrained Fuzzing*, our two main contributions.

- Lightweight Symbolic Execution is a novel approach to Symbolic Execution. Relying on an easily-enumerable constraint language, we produce approximate path predicates, which lead to a specific branch in the execution trace. While those predicates are not as precise as regular path predicates, we ensure that they are always correct-enough. In exchange for the loss of precision, we get predicates that we can solve, and even enumerate solutions for, with a linear backtrack algorithm, instead of an SMT solver. Here we consider two types of predicates: target predicates lead to an existing branch from the trace, which was deemed interesting, while inversion predicates lead to a branch was not taken by the current execution, by negating a condition in the trace.

- Constrained Fuzzing is a new type of fuzzing. While fuzzing usually creates random test cases, with the goal of maximizing coverage, constrained fuzzing creates solutions to constraints. By combining it with LSE, we use fuzzing to create multiple solutions to target predicates, so as to efficiently explore the new parts of the code the branch leads to. We also reuse the fuzzer's logging mechanism to retrieve the trace at runtime, in order to guide the Lightweight Symbolic Execution.

The resulting tool, ConFuzz, is able to explore new parts of the code by using LSE on negated conditions. This lets us craft a test case that now satisfies this condition. Furthermore, we can target branches that lead to a new part of the code. By using fuzzing to create multiple solutions that satisfy the target predicate, we efficiently explore the code below the target branch. We proved this in practice by evaluating ConFuzz on a standard fuzzing benchmark. We found it out-performed state of the art fuzzing and symbolic execution, and was also slightly better than one of the most popular combination of the two techniques.

Perspectives

To conclude, we propose three possible ways to continue the work presented in this thesis.

Constraint Language Extensions For now, our Constraint Language is limited to constraints of the form $x = y$ and $x \in [\![a; b]\!]$, as discussed in Section 4.6. We believe it is possible to increase the language's expressivity, while keeping it easily-enumerable. Here we will present two possible extensions:

sets of intervals this would allow us to represent difference constraints – making the LSE fully correct – as well as arithmetic overflows, or conditions such as **bvmod** $x\ 4 = 0$. In this situation, $c \rightsquigarrow x \in \{[\![a; b]\!]\ ;\ [\![c; d]\!]\}$ means that for the condition c to be true, x needs to be either in $[\![a; b]\!]$ or in $[\![c; d]\!]$. We could create those constraints similarly to equality analysis, or when doing a value analysis and encountering an operation that would split the values. As long as we do not have equality constraints, the main difference would be when fuzzing solutions, after the offset was modified in havoc phase. To check whether the input still satisfies the constraint, we would need to go through all intervals in the sets. As for equality constraints, they would require us to go through both sets of intervals when merging constraints from class leader and class members. That would not affect the easy-enumerability however, since normalization only happens once.

relations between variables let us imagine we have $x = y + z$ as a constraint. We would infer it by analyzing a condition's pattern, same as equality analysis. The idea for fuzzing is that anytime we update one of the variables from the relation, we update another one as well – ordered by offset index, for example. So, if we had $\{x \mapsto 6; y \mapsto 4; z \mapsto 2\}$, and mutation set z to 5, we would update x to 9. This would require some work to check whether

the language still is easy-enumerable, but it would be an interesting idea to pursue.

Exploiting the constraints As mentioned in Section 5.6, we could use the information from the constraints for more than constraint solving. We could use interval constraints as an indication of interesting values, and actively aim to enumerate values if the interval is small, or at least set it to the bounds.

Furthermore, we could introduce the concept of relevant variables: variables that are more interesting to mutate, because they have an impact on the path. For example, when targeting a branch, we could theorize that since constrained offsets were involved in previous conditions, there are more interesting to mutate than random offsets. This can be taken one step further for negated conditions. For now, we dismiss any condition for which we cannot get a value constraint. We could instead use the dependency analysis to identify which offsets are relevant to the condition. We would then mutate those to try creating a solution to the condition, even if we do not have a constraint on how we should mutate them.

Exploring synergy with full Symbolic Execution When conditions are too complicated for LSE, we cannot express them and have to resort to dependency analysis. However, this does not ensure that we will create a solution, when in invert mode. To do this, we could call regular symbolic execution. We would call Dynamic Symbolic Execution on the trace we struggle with, so as to get a perfect path predicate for the inverted branch. An SMT-solver would then be able to craft a solution, since we cannot use fuzzing to solve complex predicates. Said solution should reach a new part of the code, and can then be added to the fuzzer's database as a an interesting test case. In practice, this would require us to pick a DSE tool and an SMT solver. Furthermore, we would have to assess the cost of calling an SMT solver, compared to the gain. However, as this would only apply to a few conditions, rather than every path, we hope it might efficiently complete ConFuzz's technique.

Appendix A

Mutations List

In its mutation phase, AFL applies one of several mutations. Algorithm 4 and Figure A.1 describe the mutations, as well as how one is chosen at random during havoc phase.

Algorithm 4: AFL's mutations

Result: applies a random mutation to pos_modif

```
1  switch UR(15 + ((extras_cnt + a_extras_cnt) ? 2 : 0)) do
       // when relevant, endianness is random
2      case 0 do // flip a bit;
3      case 1 do // set byte to interesting value;
4      case 2 do // set word to interesting value;
5      case 3 do // set dword to interesting value;
       // when adding or subbing, cst is in [1..35]
6      case 4 do // randomly sub from byte;
7      case 5 do // randomly add to byte;
8      case 6 do // randomly sub from word;
9      case 7 do // randomly add to word;
10     case 8 do // randomly sub from dword;
11     case 9 do // randomly add to dword;
12     case 10 do // set to a random value;
13     case 11..12 do // delete bytes, len = choose_block_len ;
14     case 13 do // clone bytes (75%) or insert block;
15     case 14 do // overwrite bytes with random hunk (75%) or fixed
        bytes;
16     case 15 do // overwrite bytes with an extra;
17     case 16 do // insert an extra;
```

```
#define INTERESTING_8 \
  -128,          /* Overflow signed 8-bit when decremented  */ \
  -1,            /*                                         */ \
   0,            /*                                         */ \
   1,            /*                                         */ \
   16,           /* One-off with common buffer size         */ \
   32,           /* One-off with common buffer size         */ \
   64,           /* One-off with common buffer size         */ \
   100,          /* One-off with common buffer size         */ \
   127           /* Overflow signed 8-bit when incremented  */

#define INTERESTING_16 \
  -32768,        /* Overflow signed 16-bit when decremented */ \
  -129,          /* Overflow signed 8-bit                   */ \
   128,          /* Overflow signed 8-bit                   */ \
   255,          /* Overflow unsig 8-bit when incremented   */ \
   256,          /* Overflow unsig 8-bit                    */ \
   512,          /* One-off with common buffer size         */ \
   1000,         /* One-off with common buffer size         */ \
   1024,         /* One-off with common buffer size         */ \
   4096,         /* One-off with common buffer size         */ \
   32767         /* Overflow signed 16-bit when incremented */

#define INTERESTING_32 \
  -2147483648LL, /* Overflow signed 32-bit when decremented */ \
  -100663046,    /* Large negative number (endian-agnostic) */ \
  -32769,        /* Overflow signed 16-bit                  */ \
   32768,        /* Overflow signed 16-bit                  */ \
   65535,        /* Overflow unsig 16-bit when incremented  */ \
   65536,        /* Overflow unsig 16 bit                   */ \
   100663045,    /* Large positive number (endian-agnostic) */ \
   2147483647    /* Overflow signed 32-bit when incremented */

static s8  interesting_8[]  = { INTERESTING_8 };
static s16 interesting_16[] = { INTERESTING_8, INTERESTING_16 };
static s32 interesting_32[] = { INTERESTING_8, INTERESTING_16, INTERESTING_32 };
```

Figure A.1: Interesting values

Appendix B

Communication format

In this section, we present the formats used by LSE and the constrained fuzzer to communicate.

```
type afl_trace =
  { file_id : string ;
    is_seed : bool ;
    new_trans_id : int ;
    esp_val : int ;
    ebp_val : int ;
    ids_trace : int array ;
    read_sizes : int array ;
    alloc_addr : int array ;
  }
```

Figure B.1: Trace format

```
type val_cstr =
  { lo_offs : int ;
    hi_offs : int ;
    lo_val  : int ;
    hi_val  : int ;
  }

type cstr =
  { file_name : string ;
    from_inv  : bool ;
    dep_cstrs : int list ;
    uvalues   : val_cstr list ;
    svalues   : val_cstr list ;
  }
```

Figure B.2: Constraints format

Bibliography

[1] Paul Ammann and Jeff Offutt. *Introduction to Software Testing*. Cambridge University Press, USA, 1 edition, 2008.

[2] Gogul Balakrishnan and Thomas Reps. Wysinwyx: What you see is not what you execute. *ACM Trans. Program. Lang. Syst.*, 32(6), August 2010.

[3] Patrick Baudin, François Bobot, David Bühler, Loïc Correnson, Florent Kirchner, Nikolai Kosmatov, André Maroneze, Valentin Perrelle, Virgile Prevosto, Julien Signoles, and Nicky Williams. The dogged pursuit of bug-free c programs: The frama-c software analysis platform. *Commun. ACM*, 64(8):56–68, July 2021.

[4] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 2329–2344, New York, NY, USA, 2017. Association for Computing Machinery.

[5] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as markov chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1032–1043. ACM, 2016.

[6] Peter Boonstoppel, Cristian Cadar, and Dawson Engler. Rwset: Attacking path explosion in constraint-based test generation. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 351–366, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[7] Jacob Burnim and Koushik Sen. Heuristics for scalable dynamic test generation. In *23rd IEEE/ACM International Conference on Automated Software Engineering (ASE 2008), 15-19 September 2008, L'Aquila, Italy*, pages 443–446. IEEE Computer Society, 2008.

[8] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008.

[9] Cristian Cadar and Dawson Engler. Execution generated test cases: How to make systems code crash itself. In Patrice Godefroid, editor, *Model Checking Software*, pages 2–23, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[10] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. EXE: automatically generating inputs of death. In Ari Juels, Rebecca N. Wright, and Sabrina De Capitani di Vimercati, editors, *Proceedings of the 13th ACM Conference on Computer and Communications Security, CCS 2006, Alexandria, VA, USA, October 30 - November 3, 2006*, pages 322–335. ACM, 2006.

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

[12] Nuno Carvalho, Cristiano Sousa, Jorge Pinto, and Aaron Tomb. Formal verification of klibc with the wp frama-c plug-in. In *NASA Formal Methods*, volume 8430, pages 343–358, 04 2014.

[13] Hongxu Chen, Yinxing Xue, Yuekang Li, Bihuan Chen, Xiaofei Xie, Xiuheng Wu, and Yang Liu. Hawkeye: Towards a desired directed grey-box fuzzer. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 2095–2108, New York, NY, USA, 2018. Association for Computing Machinery.

[14] Peng Chen and Hao Chen. Angora: Efficient fuzzing by principled search. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 711–725. IEEE, 2018.

[15] Peng Chen, Jianzhong Liu, and Hao Chen. Matryoshka: fuzzing deeply nested branches. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 499–513, 2019.

[16] Jaeseung Choi, Joonun Jang, Choongwoo Han, and Sang Kil Cha. Grey-box concolic testing on binary code. In *Proceedings of the International Conference on Software Engineering*, pages 736–747, 2019.

[17] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In Dexter Kozen, editor, *Logics of Programs*, pages 52–71, Berlin, Heidelberg, 1982. Springer Berlin Heidelberg.

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

[19] Sylvain Conchon, Amit Goel, Sava Krstić, Alain Mebsout, and Fatiha Zaïdi. Cubicle: A parallel smt-based model checker for parameterized systems. In P. Madhusudan and Sanjit A. Seshia, editors, *Computer Aided Verification*, pages 718–724, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[20] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In Robert M. Graham, Michael A. Harrison, and Ravi Sethi, editors, *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*, pages 238–252. ACM, 1977.

[21] Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. The astreé analyzer. In Shmuel Sagiv, editor, *Programming Languages and Systems, 14th European Symposium on Programming,ESOP 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings*, volume 3444 of *Lecture Notes in Computer Science*, pages 21–30. Springer, 2005.

[22] S.R. Dalal, A. Jain, N. Karunanithi, J.M. Leaton, C.M. Lott, G.C. Patton, and B.M. Horowitz. Model-based testing in practice. In *Proceedings of the 1999 International Conference on Software Engineering (IEEE Cat. No.99CB37002)*, pages 285–294, 1999.

[23] Robin David, Sébastien Bardin, Josselin Feist, Laurent Mounier, Marie-Laure Potet, Thanh Dinh Ta, and Jean-Yves Marion. Specification of concretization and symbolization policies in symbolic execution. In Andreas Zeller and Abhik Roychoudhury, editors, *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18-20, 2016*, pages 36–46. ACM, 2016.

[24] Robin David, Sébastien Bardin, Thanh Dinh Ta, Laurent Mounier, Josselin Feist, Marie-Laure Potet, and Jean-Yves Marion. BINSEC/SE: A dynamic symbolic execution toolkit for binary-level analysis. In *IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering, SANER 2016, Suita, Osaka, Japan, March 14-18, 2016 - Volume 1*, pages 653–656. IEEE Computer Society, 2016.

[25] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8):453–457, August 1975.

[26] Adel Djoudi and Sébastien Bardin. Binsec: Binary code analysis with low-level regions. In Christel Baier and Cesare Tinelli, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 212–217, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.

[27] Adel Djoudi, Sébastien Bardin, and Éric Goubault. Recovering high-level conditions from binary programs. In John S. Fitzgerald, Constance L. Heitmeyer, Stefania Gnesi, and Anna Philippou, editors, *FM 2016: Formal Methods - 21st International Symposium, Limassol, Cyprus, November 9-11, 2016, Proceedings*, volume 9995 of *Lecture Notes in Computer Science*, pages 235–253, 2016.

[28] B. Dolan-Gavitt, P. Hulin, E. Kirda, T. Leek, A. Mambretti, W. Robertson, F. Ulrich, and R. Whelan. Lava: Large-scale automated vulnerability addition. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 110–121, May 2016.

[29] Vijay D'Silva, Daniel Kroening, and Georg Weissenbacher. A survey of automated techniques for formal software verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(7):1165–1178, 2008.

[30] Benjamin Farinier, Robin David, Sébastien Bardin, and Matthieu Lemerre. Arrays made simpler: An efficient, scalable and thorough preprocessing. In Gilles Barthe, Geoff Sutcliffe, and Margus Veanes, editors, *LPAR-22. 22nd International Conference on Logic for Programming, Artificial Intelligence and Reasoning*, volume 57 of *EPiC Series in Computing*, pages 363–380. EasyChair, 2018.

[31] Jean-Christophe Filliâtre and Andrei Paskevich. Why3 — where programs meet provers. In Matthias Felleisen and Philippa Gardner, editors, *Programming Languages and Systems*, pages 125–128, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[32] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. Afl++: Combining incremental steps of fuzzing research. In *14th {USENIX} Workshop on Offensive Technologies ({WOOT} 20)*, 2020.

[33] Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuwen Tu, Kang Li, Zhongyu Pei, and Zuoning Chen. Collafl: Path sensitive fuzzing. In *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA*, pages 679–696, 2018.

[34] Patrice Godefroid. Higher-order test generation. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, page 258–269, New York, NY, USA, 2011. Association for Computing Machinery.

[35] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: directed automated random testing. In Vivek Sarkar and Mary W. Hall, editors, *PLDI*, pages 213–223, 2005.

[36] Patrice Godefroid, Michael Y. Levin, and David A. Molnar. SAGE: whitebox fuzzing for security testing. *Commun. ACM*, 55(3):40–44, 2012.

[37] Patrice Godefroid, Hila Peleg, and Rishabh Singh. Learn&fuzz: Machine learning for input fuzzing. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, ASE 2017, page 50–59. IEEE Press, 2017.

[38] Anjana Gosain and Ganga Sharma. A survey of dynamic program analysis techniques and tools. In Suresh Chandra Satapathy, Bhabendra Narayan Biswal, Siba K. Udgata, and J.K. Mandal, editors, *Proceedings of the 3rd International Conference on Frontiers of Intelligent Computing: Theory and Applications (FICTA) 2014*, pages 113–122, Cham, 2015. Springer International Publishing.

[39] H. Huang, P. Yao, R. Wu, Q. Shi, and C. Zhang. Pangolin: Incremental hybrid fuzzing with polyhedral path abstraction. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1613–1627, Los Alamitos, CA, USA, may 2020. IEEE Computer Society.

[40] Timotej Kapus and Cristian Cadar. A segmented memory model for symbolic execution. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2019, page 774–784, New York, NY, USA, 2019. Association for Computing Machinery.

[41] James C King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

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

[43] Volodymyr Kuznetsov, Johannes Kinder, Stefan Bucur, and George Candea. Efficient state merging in symbolic execution. In *Proceedings of the 33rd*

ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, page 193–204, New York, NY, USA, 2012. Association for Computing Machinery.

[44] Shuvendu K. Lahiri and Shaz Qadeer. Back to the future: revisiting precise program verification using SMT solvers. In George C. Necula and Philip Wadler, editors, *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*, pages 171–182. ACM, 2008.

[45] JongHyup Lee, Thanassis Avgerinos, and David Brumley. TIE: principled reverse engineering of types in binary programs. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2011, San Diego, California, USA, 6th February - 9th February 2011*. The Internet Society, 2011.

[46] Caroline Lemieux and Koushik Sen. Fairfuzz: a targeted mutation strategy for increasing greybox fuzz testing coverage. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*, pages 475–485, 2018.

[47] Wenbin Li, Franck Le Gall, and Naum Spaseski. A survey on model-based testing tools for test case generation. In Vladimir Itsykson, Andre Scedrov, and Victor Zakharov, editors, *Tools and Methods of Program Analysis*, pages 77–89, Cham, 2018. Springer International Publishing.

[48] Yuekang Li, Bihuan Chen, Mahinthan Chandramohan, Shang-Wei Lin, Yang Liu, and Alwen Tiu. Steelix: program-state based binary fuzzing. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*, pages 627–637, 2017.

[49] Valentin Jean Marie Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J Schwartz, and Maverick Woo. The art, science, and engineering of fuzzing: A survey. *IEEE Transactions on Software Engineering*, 2019.

[50] J.P. Marques Silva and K.A. Sakallah. Dynamic search-space pruning techniques in path sensitization. In *31st Design Automation Conference*, pages 705–711, 1994.

[51] Xiaozhu Meng and Barton P. Miller. Binary code is not easy. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, page 24–35, New York, NY, USA, 2016. Association for Computing Machinery.

[52] Barton P. Miller, Lars Fredriksen, and Bryan So. An empirical study of the reliability of UNIX utilities. *Commun. ACM*, 33(12):32–44, 1990.

[53] Markus Müller-Olm, David A. Schmidt, and Bernhard Steffen. Model-checking: A tutorial introduction. In Agostino Cortesi and Gilberto Filé, editors, *Static Analysis, 6th International Symposium, SAS '99, Venice, Italy, September 22-24, 1999, Proceedings*, volume 1694 of *Lecture Notes in Computer Science*, pages 330–354. Springer, 1999.

[54] Manh-Dung Nguyen, Sébastien Bardin, Richard Bonichon, Roland Groz, and Matthieu Lemerre. Binary-level directed fuzzing for use-after-free vulnerabilities. In *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*, pages 47–62, San Sebastian, October 2020. USENIX Association.

[55] Hui Peng, Yan Shoshitaishvili, and Mathias Payer. T-fuzz: fuzzing by program transformation. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 697–710. IEEE, 2018.

[56] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. Vuzzer: Application-aware evolutionary fuzzing. In *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*, 2017.

[57] Thomas W. Reps, Junghee Lim, Aditya V. Thakur, Gogul Balakrishnan, and Akash Lal. There's plenty of room at the bottom: Analyzing and verifying machine code. In Tayssir Touili, Byron Cook, and Paul B. Jackson, editors, *Computer Aided Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings*, volume 6174 of *Lecture Notes in Computer Science*, pages 41–56. Springer, 2010.

[58] Koushik Sen, Darko Marinov, and Gul Agha. Cute: A concolic unit testing engine for c. *SIGSOFT Softw. Eng. Notes*, 30(5):263–272, September 2005.

[59] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Krügel, and Giovanni Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *NDSS*, 2016.

[60] Synopsis. Heartbleed's website. http://heartbleed.com/.

[61] András Vargha and Harold D. Delaney. A critique and improvement of the cl common language effect size statistics of mcgraw and wong. *Journal of Educational and Behavioral Statistics*, 25(2):101–132, 2000.

[62] Spandan Veggalam, Sanjay Rawat, Istvan Haller, and Herbert Bos. Ifuzzer: An evolutionary interpreter fuzzer using genetic programming. In Ioannis Askoxylakis, Sotiris Ioannidis, Sokratis Katsikas, and Catherine Meadows, editors, *Computer Security – ESORICS 2016*, pages 581–601, Cham, 2016. Springer International Publishing.

[63] Website. Afl vulnerability trophy case. http://lcamtuf.coredump.cx/afl/#bugs, 2019.

[64] Website. Peach. https://www.peach.tech/, 2019.

[65] Website. radamsa: a general-purpose fuzzer. https://github.com/aoh/radamsa, 2019.

[66] Website. Spike. http://www.immunitysec.com/, 2019.

[67] Wei You, Peiyuan Zong, Kai Chen, XiaoFeng Wang, Xiaojing Liao, Pan Bian, and Bin Liang. Semfuzz: Semantics-based automatic generation of proof-of-concept exploits. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 2139–2154, New York, NY, USA, 2017. Association for Computing Machinery.

[68] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. QSYM : A practical concolic execution engine tailored for hybrid fuzzing. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 745–761, Baltimore, MD, August 2018. USENIX Association.

[69] Michal Zalewski. American fuzzy lop. http://lcamtuf.coredump.cx/afl/, 2021.

ÉCOLE DOCTORALE
Sciences et technologies de l'information et de la communication (STIC)
université PARIS-SACLAY

Titre: Fuzzing et méthodes symboliques pour la détection de vulnérabilités à large échelle
Mots clés: sécurité, méthodes symboliques, fuzzing, vulnérabilités

Résumé: Alors que les programmes informatiques se répandent, le risque de bugs augmente. Dans cette thèse, nous voulons trouver d'éventuels bugs dans des programmes finis et publics.

Pour cela, nous utilisons la génération automatique de tests. Complémentant les tests écrits à la main, les générateurs de tests fabriquent automatiquement une série de tests, avec pour but de maximiser la couverture de code et de minimiser l'effort humain. Actuellement, les techniques de génération de test les plus répandues dans l'académique et dans l'industrie sont basées sur l'exécution symbolique ou le fuzzing.

- L'exécution symbolique vise à explorer complètement les chemins d'exécution. Pour cela, chaque chemin est exécuté sur une entrée symbolique, et une contrainte est inférée sur cette entrée. Quand l'analyse atteint la fin d'un chemin, cette contrainte est un prédicat de chemin, et l'exécution concrète du programme sur n'importe laquelle de ses solutions suivra le chemin voulu. De telles entrées concrètes sont générées avec un solveur de contraintes, et forment la série de tests. Cependant, il n'est pas toujours possible d'explorer tous les chemins en un temps raisonnable, et il est souvent nécessaire de borner l'exploration.

- Le fuzzing vise à exécuter le programme sur de nombreuses entrées, en espérant explorer tous les chemins possibles. Il dépend donc d'une génération d'entrées rapide et facile. Tandis que les fuzzers de base fonctionnent en boîte noire et génèrent des entrées aléatoires indépendamment du programme, les fuzzers en boîte grise utilisent une analyse pour obtenir des informations à propos du programme. Ces informations sont alors utilisées pour améliorer la génération d'entrées. Cependant, malgré ces améliorations, les fuzzers ont toujours de la difficulté à trouver la solution de conditions qui ont une faible probabilité d'être vraies.

Ainsi, l'exécution symbolique et le fuzzing exhibent des forces et des faiblesses complémentaires, nous poussant à les combiner.

Pendant cette thèse, nous avons développé une technique de génération de test automatisée, qui combine la puissance de raisonnement de l'exécution symbolique pour s'attaquer au code complexe, et le faible coût du fuzzing pour générer des entrées efficacement.

La solution que nous proposons combine deux nouvelles idées: l'Exécution Symbolique Légère (ESL) et le Fuzzing Contraint. L'Exécution Symbolique Légère est une variante de l'exécution symbolique où l'analyse s'arrête sur une condition d'un chemin, plutôt qu'à la fin, et le langage de contraintes ciblé est réduit à un fragment facilement énumérable de l'habituel. Par conséquent, dériver des prédicats de chemin (corrects) dans ce langage est plus compliqué, mais il est facile d'énumérer des entrées exerçant un chemin, sans utiliser de solveur de contraintes. Deuxièmement, le Fuzzer Contraint manipule une entrée et une contrainte facilement énumérable, et génère de nouvelles entrées qui satisfont la contrainte et suivent donc le chemin, jusqu'à la condition ciblée. En général, l'ESL guidera l'exploration au-delà des conditions difficiles et vers les parties intéressantes du code, tandis que le fuzzer contraint créera efficacement des entrées, y compris des solutions aux contraintes. Cela nous permet d'explorer le programme sans systématiquement faire appel à l'analyse symbolique, et supprime la dépendance à un solveur pour créer des entrées satisfaisant les contraintes.

Nous avons évalué les performances de l'outil utilisant ces techniques, appelé ConFuzz, sur un banc de tests standard du fuzzing. La conclusion de cette expérimentation est que ConFuzz a de meilleurs performances que l'état de l'art du fuzzing et de l'exécution symbolique.

Title: Software security: combining fuzzing and symbolic methods for vulnerability detection
Keywords: symbolic methods, fuzzing, security, vulnerabilities

Abstract: As computer programs spread, the risk of bugs increases. In this thesis, we want to find possible bugs in finished and released programs.

We do this through automatic test generation, a major topic in software engineering and security. A complement to hand-crafted tests, test generators automatically build test suites, aiming to maximize program coverage and minimize human effort. Currently, most test generation techniques and tools studied by researchers and applied in industry rely on some form of either symbolic execution or fuzzing.

- Symbolic execution aim to exhaustively explore the possible execution paths. It achieves this by executing each path on a symbolic input, and inferring a constraint on said input. When the analysis reaches the end of a path, this constraint is a path predicate, and a concrete execution of the program on any of its solution will follow the intended path. Such test cases are generated using an off-the-shelf solver, and form the test suite. However, it is not always possible to explore all paths in reasonable time, and we often have to bound the exploration.

- Fuzzing aims to run the program on many test cases, in order to hopefully trigger all possible paths. As such, it relies on quick and easy test case generation. While the most basic fuzzers function in a blackbox manner and generate random test cases independently from the program, greybox fuzzers also rely on an analysis to gain some information about the program. This information is then used to make the test case generation more efficient. However, despite this improvement, fuzzers still struggle with finding the solution to conditions that have a low probability of being true, such as password checks.

Hence, symbolic execution and fuzzing exhibit rather complementary strengths and weaknesses, calling for a proper integration between the two techniques.

During this thesis, we developed an automated test generation technique, combining the reasoning power of symbolic execution to tackle complex code with the light cost of greybox fuzzing to generate test cases efficiently.

The solution we propose combines two novel ideas: Lightweight Symbolic Execution (LSE) and Constrained Fuzzing. Lightweight Symbolic Execution is a variant of symbolic execution where the analysis targets a condition on a path, rather than a full path, and the target constraint language is restricted to an easily-enumerable fragment of the usual one. As a consequence, deriving (correct) path predicates in this language is more complicated but test cases following a given path are then easy to enumerate, without using any off-the-shelf constraint solver. Second, a Constrained Fuzzer operates over a test case and an easily-enumerable constraint in order to quickly generate test cases which follow the intended path, up to the targeted condition. Overall, LSE will lead the exploration past difficult conditions and towards interesting parts of the code, while the constrained fuzzer will efficiently create test cases, including solutions to the constraints. This allows us to explore the program without systematically relying on symbolic analysis, and removes the need for an SMT solver to create test cases satisfying the constraints.

We evaluated the performances of the resulting tool, called ConFuzz, on a standard fuzzing benchmark, and found that we improved upon the performance of standard fuzzing and symbolic execution.