Increasing the performance of symbolic execution by compiling symbolic handling into binaries

Sebastian Poeplau

INCREASING THE PERFORMANCE OF SYMBOLIC EXECUTION BY COMPILING SYMBOLIC HANDLING INTO BINARIES

SEBASTIAN POEPLAU

Sorbonne Université
Ecole Doctorale Informatique, Télécommunications et Electronique (ED130)

Thèse de doctorat d'Informatique
dirigée par Aurélien Francillon

Présentée et soutenue publiquement le 20 novembre 2020
devant un jury composé de :

Prof. Cristian Cadar	(rapporteur)	Imperial College London
Dr. Tamara Rezk	(rapporteuse)	INRIA Sophia Antipolis-Méditerranée
Prof. Aurélien Francillon	(directeur de thèse)	EURECOM
Dr. Christophe Hauser		ISI, University of Southern California
Dr. Sarah Zennou		Airbus
Prof. Davide Balzarotti		EURECOM
Dr. Khaled Yakdan	(invité)	Code Intelligence

For Chrissie,
for my parents,
and for Duda

ABSTRACT

Symbolic execution has the potential to make software more secure by significantly improving automated vulnerability search. Its principled reasoning can automatically explore parts of the program under test that would otherwise be hard to reach. However, many current implementations face challenges in scalability and usability: The power of symbolic execution comes at a cost, and the community is exploring different approaches to make symbolic executors both efficient and easy to use.

In this thesis, we propose a general technique that allows for more efficient implementations of the execution component in symbolic executors. We first examine the state of the art and analyze the strengths and weaknesses of current systems. From the results of this comparison, we derive the idea of accelerating execution by embedding symbolic handling into compiled programs instead of symbolically interpreting a higher-level representation of the program under test. Using this approach, we develop a compiler-based symbolic executor and show that it indeed achieves high execution speed in comparison with state-of-the-art systems. Since the compiler is limited to scenarios where source code of the program under test is available, we then design and implement a complementary solution for symbolic execution of binaries; it uses the same basic idea of embedding symbolic execution into fast machine code but combines the approach with binary translation to handle the challenges of binary-only analysis. Both systems, the compiler-based symbolic executor as well as the binary translator, put a strong focus on ease of use with the goal of emphasizing that symbolic execution can benefit software developers and analysts in various fields. We conclude by discussing research directions that could lead to even more practical systems and ultimately enable the use of symbolic execution in mainstream software testing.

RÉSUMÉ

L'exécution symbolique a le potentiel de rendre les logiciels plus sûrs en améliorant considérablement la recherche automatisée des vulnérabilités. Son principe de fonctionnement permet d'explorer formellement et automatiquement toutes les parties du programme qui seraient autrement difficiles à explorer. Toutefois, de nombreuses implémentations actuelles sont confrontées à des défis en termes d'évolutivité et d'ergonomie : La puissance de l'exécution symbolique a un coût, et la communauté explore différentes approches pour rendre les exécuteurs symboliques à la fois efficaces et faciles à utiliser.

Dans la présente thèse, nous proposons une technique générale qui permet des implémentations plus efficaces de la composante d'exécution dans les exécuteurs symboliques. Nous examinons d'abord l'état de l'art et analysons les forces et les faiblesses des systèmes actuels. A partir des résultats de cette comparaison, nous dérivons l'idée d'accélérer l'exécution en intégrant la manipulation symbolique dans les programmes compilés au lieu d'interpréter symboliquement une représentation de plus haut niveau du programme testé. En utilisant cette approche, nous développons un exécuteur symbolique basé sur le compilateur et nous montrons qu'il atteint effectivement une vitesse d'exécution élevée par rapport aux systèmes actuellement les plus performants. Comme le compilateur est limité aux scénarios où le code source du programme testé est disponible, nous concevons et mettons en œuvre une solution complémentaire pour l'exécution symbolique des binaires ; elle utilise la même idée de base d'intégration de l'exécution symbolique dans un code machine rapide, mais combine l'approche avec la traduction binaire pour relever les défis de l'analyse binaire seule. Les deux systèmes, l'exécuteur symbolique basé sur le compilateur ainsi que le traducteur binaire, mettent fortement l'accent sur la facilité d'utilisation dans le but de souligner que l'exécution symbolique peut profiter aux développeurs et aux analystes de logiciels dans divers domaines. Nous concluons en discutant des directions de recherche qui pourraient conduire à des systèmes encore plus pratiques et permettre en fin de compte l'utilisation de l'exécution symbolique dans les tests de logiciels courants.

PUBLICATIONS

The thesis is based on three articles:

- Sebastian Poeplau and Aurélien Francillon. "Systematic comparison of symbolic execution systems: Intermediate representation and its generation." In: *Proceedings of the 35th Annual Computer Security Applications Conference*. ACM. 2019, pp. 163–176.

  This publication forms the basis of Chapter 3, and parts of it are used in Chapters 1 and 2.

- Sebastian Poeplau and Aurélien Francillon. "Symbolic execution with SymCC: Don't interpret, compile!" In: *29th USENIX Security Symposium (USENIX Security 20)*. Distinguished Paper Award. Boston, MA: USENIX Association, Aug. 2020. URL: https://www.usenix.org/conference/usenixsecurity20/presentation/poeplau.

  Chapter 4 is based on this article, and Chapters 1 and 2 use some of its material.

- Sebastian Poeplau and Aurélien Francillon. "SymQEMU: Compilation-based symbolic execution for binaries." Under submission. Sept. 2020.

  This is the basis of Chapter 5, and Chapter 2 contains parts of it.

Apart from those, I have been involved in the following publications during my doctoral studies:

- Giovanni Camurati, Sebastian Poeplau, Marius Muench, Tom Hayes, and Aurélien Francillon. "Screaming Channels: When Electromagnetic Side Channels Meet Radio Transceivers." In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM. 2018, pp. 163–177.

- Nassim Corteggiani, Giovanni Camurati, Marius Muench, Sebastian Poeplau, and Aurélien Francillon. "SoC Security Evaluation: Reflections on Methodology and Tooling." In: *IEEE Design & Test* (2020).

ACKNOWLEDGMENTS

Obtaining a PhD is a nontrivial undertaking, and I am grateful for all the support that I received on the way.

First and foremost, I have to thank my amazing wife Christine for supporting my back-and-forth travels between France and Germany for the past three years. She encouraged me whenever I needed encouragement, and she cheered with me when there was reason to celebrate. Moreover, I am extremely grateful to my parents and my sister, without whose tireless support I could not have managed.

I consider myself very lucky to have encountered the most welcoming environment at EURECOM: I am deeply grateful to my PhD advisor Aurélien Francillon for teaching me how to do research while giving me all the freedom I wanted, and to the amazing S3 group—a group of friends rather than colleagues. I greatly enjoyed spending time with all of you!

Last but not least, several people and organizations helped with my research. In particular, I am grateful to Insu Yun for patiently answering lots of questions about QSYM, and I would like to thank Vitaly Chipounov for his help with S2E. Code Intelligence provided me with valuable insights into industrial applications of fuzzing, and some of the experiments described in this thesis were carried out using the Grid'5000 testbed, a cluster of powerful machines that we were graciously allowed to use for free. Finally, my research was supported by the DAPCODS/IOTics ANR 2016 project (ANR-16-CE25-0015).

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF LISTINGS

1 INTRODUCTION

Information technology did not exist a mere century ago, but we have come to rely on it heavily in the past few decades. Most modern businesses would be unimaginable without computer-aided means of data storage and processing. Our everyday lives are impacted heavily by technology, and we rely on it in a plethora of ways, in areas as diverse as communication, transportation and health.

Software is a key component in a lot of the technology we use, and thus it plays a more and more important role in modern society. Software controls the logistics underpinning our economy, it enables communication across the globe, and it quite literally drives our cars. Consequently, software failures can have great impact on our lives. While most of them are just minor annoyances, if at all, some have catastrophic consequences.

In response to the growing importance of software, our field has developed a variety of techniques to assert software correctness, ranging from static and formal approaches like model checking to dynamic and heuristic mechanisms such as fuzz testing. Independently of the concrete approach, all those techniques share the common goal of helping to build more correct software. My particular interest in the spectrum of software testing methodologies is in symbolic execution. In my perception, it unites the beauty of rigorous mathematical reasoning with the practicality of dynamic techniques.

Symbolic execution was conceived more than 40 years ago to aid in software testing [43]. While it was rather impractical initially, great advances in the field of computer-aided reasoning, in particular SAT and SMT solving, led to the first more or less practical implementations in the early 2000s [12, 13]. Since then, symbolic execution has been the subject of much research in both the software security and the verification communities [17, 74, 77, 86], and the technique has established its place in vulnerability search and program testing. In the 2016 DARPA Cyber Grand Challenge, a US-government competition in automated vulnerability finding, exploiting and fixing, symbolic execution was an integral part in the approaches of all three winning teams [15, 57, 74].

Conceptually, a symbolic execution engine keeps track of how each intermediate value is computed while executing a program. Whenever the program hits a symbolic conditional—i.e., a branch whose outcome depends on the program input—the symbolic execution engine can pass the collected information to a solver in order to generate new inputs that yield the desired outcome at the branch point. In other

words, symbolic execution can ideally generate exactly one input for each possible path through the program under test.

Recent years have seen the development of several symbolic execution engines, both in academic environments and by commercial actors [3]. However, the performance of symbolic execution remains a major challenge, especially when the technique is applied to larger software systems. Recent work has shown that combining symbolic execution with fuzz testing has the potential of handling the weaknesses of either approach and combining their strengths [77, 86]. In this context, the speed of symbolic execution is of the essence: Exploration is driven by the fuzzer, which also takes care of vulnerability checks by leveraging sanitizers, and the main task of the symbolic execution engine is to generate relevant new test inputs as quickly as possible.

The overarching goal of my work during the PhD program is to enable faster symbolic execution, leading to more efficient program exploration, and ultimately enabling easier discovery of software defects. To this end, we first studied the performance of popular symbolic execution engines, trying to discern design aspects that play an important role in the speed of symbolic execution. Based on the insights from this study, we developed a novel approach to symbolic execution that significantly improves execution speed by using a compiler where previous implementations would employ an interpreter. Since our technique relies on the availability of source code for the program under test, the third step of my research consisted in the development of a similarly fast symbolic executor for binary-only scenarios. Each of the three projects has been described in a peer-reviewed article (with the last one still under submission at the time of this writing); the thesis is based on material from those three articles, rearranged, enhanced and modified where necessary to yield a cohesive narrative.

The thesis is structured as follows: We start by presenting relevant background information (Chapter 2). Next, we describe our performance comparison of existing symbolic execution systems (Chapter 3). It inspired the concept of compilation-based symbolic execution (Chapter 4). We then extended the reach of our novel approach to binaries with no source code available (Chapter 5). Finally, we discuss future work and conclude the thesis (Chapter 6). We present related work per chapter in order to be able to frame it in the appropriate context.

2 BACKGROUND

This chapter discusses background information that is relevant to the rest of the thesis.

2.1 SYMBOLIC EXECUTION

Symbolic execution was originally proposed by King in 1975 [43]. It was envisioned as a technique for software testing that is more rigorous than manual tests and more practical than formal verification. The early 2000s have finally seen the development of several more or less practical symbolic execution engines [13], fueled by significant improvements in *Boolean satisfiability (SAT)* and *satisfiability modulo theories (SMT)* solving [89], and the field continues to be very active to this day. Modern implementations are typically designed to answer questions like "can this array access run out of bounds," "is it possible to take this branch of the program," or "can this pointer be null when it is dereferenced?" Moreover, if the answer is affirmative, symbolic executors typically provide a test case, i.e., a new program input that triggers the requested behavior. This ability makes symbolic execution extremely useful for automated program testing, where the goal is to explore as many corner cases of a program as possible and find inputs that cause crashes or otherwise trigger bugs.

At the core of most modern symbolic execution engines, an interpreter runs the program under test while keeping a record of how each intermediate value in the program is computed. Those computations are typically expressed in the logic of bit vectors and arrays. A noteworthy exception is QSYM [86], to be discussed in more detail later, which executes x86 machine code directly. Whenever the target program encounters a conditional whose outcome depends on intermediate values, the symbolic execution engine can express the condition in terms of the original input values of the program, using the knowledge of how the intermediate values were derived from the inputs in the course of execution. Subsequently, the system needs to solve the so-called *path constraints* for input values in order to generate new inputs that cause the program to run up to the conditional and then take the desired path out of it; in other words, the symbolic executor needs to solve formulas in the logic of bitvectors and arrays (see Section 3.4.5 for examples). The field of SMT solving provides tools to address this (generally hard [45]) problem: in many cases, modern SMT solvers can solve such difficult queries in acceptable time, using various heuristics that are themselves an active area of

research. It is, however, in the best interest of any symbolic execution engine to generate queries in a form that SMT solvers can solve quickly. In Chapter 3 we show that the way IR is generated has a profound impact on the complexity of the resulting SMT queries.

When symbolic execution is used with the goal of testing an entire program, the execution engine typically tries to follow each path out of any conditional statement, i.e., it *forks* and tries to generate inputs for each possible outcome. A common problem arising from forking at each conditional is *path explosion*: the number of paths to explore grows exponentially over time. More recent approaches combine symbolic execution with fast random testing [77, 86]. In this latter scenario, a fuzzer selects interesting inputs, and symbolic execution merely follows a fixed path dictated by a given concrete program input—a process called *concolic execution*. The symbolic execution engine thus does not have to cope with path explosion; it just uses the solver to compute inputs that diverge from the predetermined path at any desired point, possibly even trading precision for speed [86]. Especially in this hybrid setting, faster symbolic execution amounts to more tested code and—all else being equal—a higher chance of detecting vulnerabilities.

Note that any concolic executor can be converted into an exhaustive explorer by wrapping it in a loop that repeatedly feeds newly generated inputs back to the symbolic executor (see Listing 2.1). Provided that the system keeps track of previously encountered paths across executions, the process terminates once the target program has been fully explored.

Listing 2.1: Shell script that transforms a concolic executor into a simple exhaustive explorer, loosely modeled after SAGE [33]. The executor is assumed to read the test case to follow from standard input and generate new test cases as files in a directory called `output`; the initial set of test cases is expected in the directory `next_gen`.

```
while ls -A $next_gen; do
    mv next_gen cur_gen
    for f in cur_gen/*; do
        concolic_executor < $f
        for new_case in output/*; do
            mv $new_case next_gen/$(sha256sum $new_case)
        done
    done
done
```

In summary, every implementation of symbolic execution is constructed from a set of basic building blocks (see Figure 2.1):

EXECUTION The program under test is executed, and the system produces symbolic expressions representing the computations.

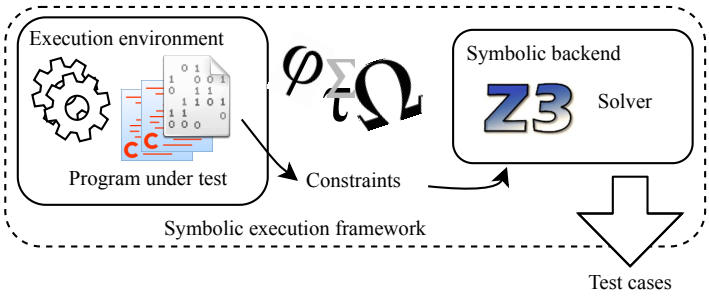

Figure 2.1: The building blocks of symbolic execution. The entire system may be encapsulated in a component that handles forking and scheduling.

These expressions are the essential asset for reasoning about the program.

SYMBOLIC BACKEND The sole purpose of describing computations symbolically is to reason about them, e.g., to generate new program inputs that trigger a certain security vulnerability. The symbolic backend comprises the components that are involved in the reasoning process. Typically, implementations use an SMT solver, possibly enhanced by pre-processing techniques. For example, KLEE [12] employs elaborate caching mechanisms to minimize the number of solver queries, and QSYM [86] removes all irrelevant information from queries to reduce the load on the solver.

FORKING AND SCHEDULING Some implementations execute the target program only a single time, possibly along the path dictated by a given program input, and generate new program inputs based on that single execution (e.g., SAGE [33], Driller [77] and QSYM [86]). On the other hand, several other implementations contain additional facilities to manage multiple executions of the program under test along different paths by forking execution at branch points in the program (e.g., KLEE [12], Mayhem [15] and angr [74]).

The three building blocks—execution, symbolic backend, and forking and scheduling—are conceptually orthogonal to each other, even if implementations sometimes lack a clear distinction. In Chapters 4 and 5, we show that accelerating the execution component yields significant improvements in the overall performance of symbolic execution engines. We discuss possible future work on the other components in Chapter 6.

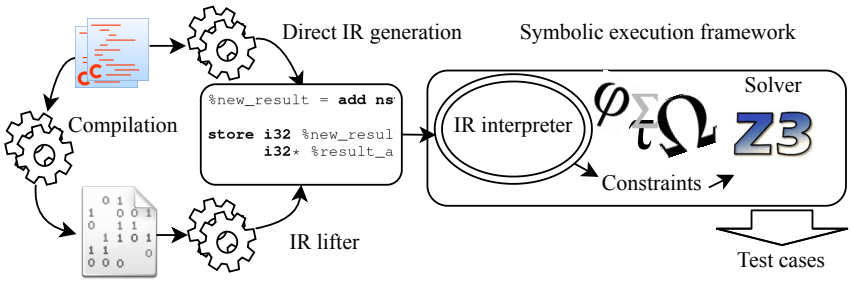

Figure 2.2: IR-based symbolic execution interprets IR and interacts with the symbolic backend at the same time.

2.2 INTERMEDIATE REPRESENTATION

When emulating the execution of a program, symbolic execution faces the challenge that the instruction sets of modern CPUs are large and complex; writing a symbolic emulator for them is not trivial. Therefore, it is common to *lift* the program under test to some intermediate representation, which is then emulated. Symbolic execution at the IR level also increases portability: in order to support a new architecture, one "only" needs to reimplement the IR generator, while the rest of the system can remain unchanged.

Symbolic execution engines differ in the choice of IR and in their approach to generating IR from either a binary or from source code. We refer to the process as *IR generation*, no matter whether the initial artifact is a machine-code binary or source code, because the term *lifting* is only appropriate for IR generation that starts from machine code. The choice of IR-generation mechanism has considerable impact on several aspects of symbolic execution (see Chapter 3).

2.2.1 *IR-based symbolic execution*

A common way of implementing symbolic execution is by means of an *intermediate representation (IR)*. Compared to the native instruction sets of popular CPU architectures, IRs typically describe program behavior at a high level and with fewer instructions. It is therefore much easier to implement a symbolic interpreter for IRs than for machine code directly, so this is the approach that many state-of-the-art systems take.

IR-based symbolic execution first needs to transform the program under analysis into IR. KLEE [12], for example, works on LLVM bitcode and uses the `clang` compiler to generate it from source code; S2E [17] also interprets LLVM bitcode but generates it dynamically from QEMU's internal program representation, translating each basic block as it is encountered during execution; angr [74] transforms machine code to VEX, the IR of the Valgrind framework [56]. In general, IR generation can require a significant amount of work [21],

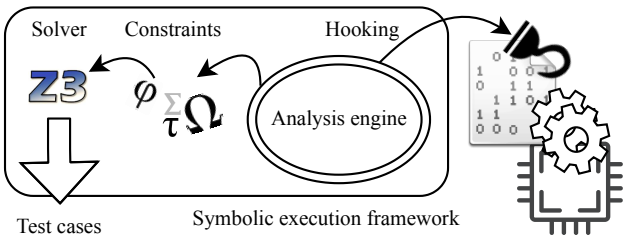

Figure 2.3: IR-less symbolic execution attaches to the machine code executing on the CPU and instruments it at run time.

especially when it starts from machine code [42]. Once the IR of the target program is available, a symbolic interpreter can run it and produce symbolic expressions corresponding to each computation. The expressions are typically passed to the symbolic backend for further processing as discussed above; Figure 2.2 illustrates the process.

2.2.2 *IR-less symbolic execution*

While translating target programs to an intermediate representation simplifies the implementation of symbolic execution, *interpreting* IR is much slower than native execution of the corresponding binary, especially in the absence of symbolic data (i.e., when no symbolic reasoning is necessary). This observation has led to the development of Triton [69] and QSYM [86], which follow a different approach: instead of translating the program under test to IR and then interpreting it, they execute the unmodified machine code and instrument it at run time. Concretely, Triton and QSYM both control the target program's execution with Intel Pin [52], a framework for binary instrumentation. Pin provides facilities for inserting custom code when certain machine-code instructions are executed. The symbolic executors use this mechanism to inject code that handles computations symbolically in addition to the concrete computations performed by the CPU. For example, when the CPU is about to add the values contained in two registers, Pin calls out to the symbolic executor, which obtains the symbolic expressions corresponding to the registers' values, produces the expression that describes the sum, and associates it with the register that receives the result of the computation. See Figure 2.3 for an overview.

The main advantage and original goal of the IR-less approach is speed. Run-time instrumentation still introduces overhead, but tracing native execution while inserting bits of code is much faster than interpreting IR. Another, more subtle, advantage is robustness: If an IR-based system does not know how to handle a certain instruction or a call to some library function, it is not able to continue because the interpreter cannot execute the requested computation; in IR-less symbolic execution, however, the CPU can always execute the target

program concretely. The injected analysis code will just fail to produce an appropriate symbolic expression. One might say that performance degrades more gracefully than in IR-based systems.

However, building symbolic execution directly on machine code has considerable downsides. Most notably, the implementation needs to handle a much larger instruction set: while the IRs that are commonly used for symbolic execution comprise a few dozen different instructions, CPU instruction sets can easily reach hundreds to thousands of them. The symbolic executor has to know how to express the semantics of each of those instructions symbolically, which results in a much more complex implementation. Another problem is architecture dependence: naturally, instrumentation of machine code is a machine-dependent endeavor. IRs, on the other hand, are usually architecture agnostic. IR-based systems therefore work on any architecture where there is a translator from the respective machine code to IR. This is especially relevant for the domain of embedded devices, where a great variety of CPU architectures is in common use.

The alternative extreme—instrumenting at the source-code level—is less common these days [13, 32, 71] and has similar downsides to the machine-code approach, with architecture-dependence replaced by dependence on a particular programming language. SymCC and SymQEMU, the systems we present in Chapters 4 and 5, respectively, use IR and thus retain the flexibility and implementation simplicity associated with IR-based approaches, yet our compilation-based technique allows them to reach (and surpass) the high performance of IR-less systems.

2.2.3 *Reducing overhead*

In either type of symbolic execution, IR-based and IR-less, building symbolic expressions and passing them to the symbolic backend is necessary only when computations involve symbolic data. Otherwise, the result is completely independent of user input and is thus irrelevant for whatever reasoning is performed in the backend. A common optimization strategy is therefore to restrict symbolic handling to computations on symbolic data and resort to a faster execution mechanism otherwise, a strategy that we call *concreteness checks*. In IR-based implementations, symbolic interpretation of IR may even alternate with native execution of machine code on the real or a fast emulated CPU; angr [74], for example, follows this approach. Implementations vary in the scope of their concreteness checks—while QSYM [86] decides whether to invoke the symbolic backend on a per-instruction basis, angr [74] places hooks on relevant operations such as memory and register accesses. Falling back to a fast execution scheme as often as possible is an important optimization, which we also implement in SymCC and SymQEMU (see Sections 4.3 and 5.4, respectively).

2.3 BINARY-ONLY SYMBOLIC EXECUTION

In some scenarios, the source code of the program under test is not available. Symbolic execution then has to function with only the binary. Requiring the analysis system to work with just a binary target adds its own unique set of challenges to the field: In the absence of source code, translating programs to an intermediate representation requires reliable disassemblers; due to the challenges of static disassembly [60], most implementations perform the translation on demand at run time [17, 74]. Moreover, support for multiple architectures becomes crucial when source code is not available: without source code, cross-compiling a program for whichever architecture a symbolic executor supports is not an option. If a symbolic execution system cannot handle the target architecture of the program under test, it simply cannot be used. This is a particular challenge for the embedded space with its large variety of processor architectures.

IR-less symbolic executors thus face severe portability challenges in the binary-only scenario, in addition to maintainability issues arising from the relatively complex implementation. Executors that translate the target program to an intermediate representation fare better, but they still require a reliable translator for the particular target architecture; significant amounts of work have gone into verifying translator correctness [42]. This is in contrast to source-based symbolic execution, where intermediate representations can rather easily be obtained from the program's source code [12].

In summary, binary-only symbolic execution puts higher demands on architectural flexibility and the performance of (run-time) program translation than source-based analysis.

3

THE PERFORMANCE IMPACT OF INTERMEDIATE REPRESENTATION AND ITS GENERATION

Most implementations of symbolic execution transform the program under analysis to some *intermediate representation (IR)*, which is then used as a basis for symbolic execution. There is a multitude of available IRs, and even more approaches to transform target programs into a respective IR.

When developing a symbolic execution engine, one needs to choose an IR, but it is not clear which influence the IR generation process has on the resulting system. What are the respective benefits for symbolic execution of generating IR from source code versus lifting machine code? Does the distinction even matter? What is the impact of not using an IR, executing machine code directly? We felt that there was little scientific evidence backing the answers to those questions. Therefore, we developed a methodology for systematic comparison of different approaches to symbolic execution; we then used it to evaluate the impact of the choice of IR and IR generation. Our comparison framework is available to the community for future research.

3.1 INTRODUCTION

Typically, symbolic execution engines translate the program under test to an *intermediate representation (IR)* which they can subsequently execute symbolically. Generating the IR from machine code may be the only solution when source code is not available. Testing the binary directly also has the advantage of testing the "shipped" product, independently of source language and compiler [8]. However, when source is available, both approaches are possible and the choice of how to generate IR is a distinguishing factor between the various approaches. There is quite some conventional wisdom surrounding it: one intuition is that high level source code semantics (e.g., buffer boundaries, types) can be preserved and will make symbolic execution, and bug finding, more efficient [21]. However, to the best of our knowledge, there was no systematic study backing such claims. The goal of this chapter is therefore to systematically assess how the choice of IR, and the process of generating it, influence various aspects of symbolic execution.

We selected several popular implementations, each with their own mechanism for IR generation, and compared them to discern the effect on their relative performance. In particular, we answer the following research questions:

1. Is there a benefit in generating IR from source code as compared to IR generation from binaries?

2. Are there significant differences between symbolic execution of different IRs generated from the same programs? What about the special case of symbolically executing machine code directly?

Along the way, we discovered that the presumably simple engineering task of setting up a number of symbolic execution engines in a stable environment and running a fair comparison on them is actually quite a challenge in itself. We therefore made our environment and dataset publicly available.

The answers to the research questions, as well as our observation of usability issues in existing systems, informed the design choices we made in our own implementations of symbolic execution, to be discussed in Chapters 4 and 5.

In summary, the contributions of this chapter are the following:

- We devise a framework for systematic comparison of different implementations of symbolic execution.

- We provide an assessment of the impact that the choice of IR generation mechanism has on the performance of a symbolic execution engine and derive recommendations for future work in symbolic execution.

3.2 DESIGN SPACE

While this study focuses on the generation of intermediate representations and the impact of that choice on the overall performance of symbolic execution, the design of a symbolic execution engine involves many other decisions. In this section, we give an overview of important dimensions in the design space and frame our particular object of study, namely the IR generation process, in the larger context. We refer interested readers to the survey by Baldoni et al. [3] for a more comprehensive discussion of symbolic execution techniques in general.

Figure 3.1 gives an overview of the components in a typical symbolic execution engine. We focus on IR and its execution, which is the core part that is present in every such system. There may be additional components, such as security checks and a machinery for state forking and scheduling. However, they are dropped in more recent symbolic execution engines, where symbolic execution functions in concert with a fuzzer which takes care of crash detection and input prioritization [77, 86].

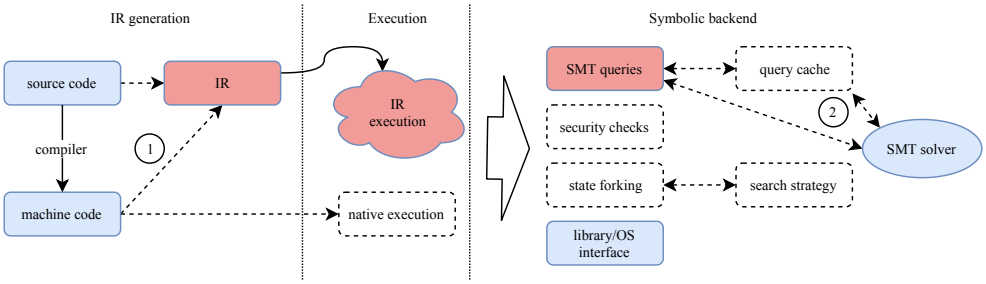

Figure 3.1: Overview of symbolic execution, showing our focus on IR, IR execution, and SMT queries. Numbers indicate orthogonal studies by (1) Kim et al. [42] and (2) Palikareva and Cadar [59] as well as Liu et al. [51]. Dashed elements are not always present. IR and machine code may be identical (e.g., in QSYM [86]).

3.2.1 *Path selection*

At each branching point in the program under analysis, a symbolic execution engine faces the decision which path to follow. In King's original proposal, the user was prompted every time [43]. Modern systems typically employ heuristics that do not rely on user interaction. There are two major approaches:

1. *Concolic execution* follows the path dictated by a given concrete input, typically generating new inputs along the way that leave the predetermined path. In this case, the question of path selection is addressed externally; it mostly revolves around choosing a concrete input to process in each iteration of the system. Examples of symbolic execution engines that follow this approach are SAGE [33] and the symbolic components of Driller [77] and QSYM [86].

2. Some symbolic execution engines choose to pursue *all* feasible code paths simultaneously, conceptually forking the executor at each branching point. The scheduling of the resulting *execution states* is a crucial element of those systems' design because a good selection strategy may quickly guide execution toward interesting code, while less sophisticated strategies risk getting stuck (e.g., in loops). KLEE [12] and Mayhem [15] are examples of symbolic execution engines that conceptually follow all code paths at once.

While the path selection strategy is crucial for the effectiveness of conventional symbolic execution, it is irrelevant for systems that run symbolic execution in concolic mode along with a fuzzer. Therefore, we use concolic mode for all systems in our study, implementing it where necessary. Concolic execution allows us to pass the same fixed input to all engines and trust that they follow the same code path.

3.2.2 *Incremental solving*

As symbolic execution follows a path through the code under analysis, it collects the constraints imposed on symbolic data at each branching point. The resulting *path constraints* are used whenever a branching point is encountered: execution may proceed down a path if and only if there exists a concrete value for the symbolic data that fulfills (1) all path constraints conjoined with (2) the desired outcome of the branching condition; the latter is subsequently added to the path constraints. Intuitively, the consequence is that path constraints are large conjunctions that build up incrementally, one conjunct per branching point in the program. Modern SMT solvers can take advantage of the incremental nature of resulting SMT queries, conceptually reusing knowledge gained in answering previous queries when processing the next increment. Liu et al. showed that incremental solving indeed leads to significant performance improvements in practice [51].

Some symbolic execution engines choose not to use incremental solving. When evaluating query complexity in our study, we therefore reset the SMT solver before each query, essentially preventing it from exploiting any incremental nature in the queries. This eliminates differences unrelated to our subject of study, which would otherwise skew the results.

3.2.3 *Interleaved execution*

Symbolic execution is only necessary when the executed code works with symbolic data—when everything is concrete, the code can as well be executed natively, which is usually significantly faster. Therefore, many symbolic execution engines have support for alternating back and forth between symbolic execution and some form of direct execution for code that does not work with symbolic data. For instance, QSYM distinguishes at the instruction level whether the code to be executed has symbolic inputs. It then only instruments instructions that need to handle symbolic data by adding complementary symbolic computations [86]. The approaches taken by the different symbolic execution engines vary in granularity and concrete execution mechanism, but they share the common goal of using fast execution techniques as often as possible and only falling back to slow symbolic execution when necessary. Therefore, even slow symbolic executors may achieve a high overall performance in terms of test coverage per time if they manage to execute a large portion of the code under test natively.

Among the systems in our study, some allow the user to configure whether or not code with only concrete data is executed natively, whereas others do not work without interleaved concrete execution or do not support it at all. We take great care to compare only results obtained using similar strategies when measurements are affected by

	KLEE	S2E	angr	QSYM
Version	`4efd7f6`	2018-09-24	7.8.8.1	`6f00c3d`
IR	LLVM	LLVM	VEX	x86
IR generator	Clang (source)	QEMU + lifter	libvex	n/a
Solver(s)	Z3 (4.4.1), others	Z3 (4.7.1)	Z3 (4.5.1)	Z3 (4.5.0)
Language	C++	C, C++	Python	C++
Concrete execution	n/a	QEMU/KVM	Unicorn	CPU

Table 3.1: Comparison of design choices relevant to our study in the four symbolic execution engines that we analyze.

this aspect of symbolic execution. We discuss this problem in more detail in Section 3.4.4.

There are many more degrees of freedom in the design of a symbolic execution system, such as the approach to state forking, query caching techniques, and vulnerability detection mechanisms. However, since we focus on concolic execution (e.g., in concert with a fuzzer), those factors do not impact our experimental setup. Therefore, we do not discuss them here and refer to the literature for details [3].

3.3 APPROACHES UNDER ANALYSIS

In this study, we compare common IR generation approaches, each represented by a tool that implements the approach. Our test set includes KLEE [12] for source-based IR generation, S2E [17] for binary-based generation, angr [74] as a binary-based approach with a different IR, and QSYM [86] as representative for systems that do not use IR at all. This section presents each of the tools, before the next section details the actual analysis. Unless otherwise noted, when talking about *machine code* we refer to the x86 and AMD64 instruction sets.

3.3.1 *KLEE*

Published in 2008, KLEE [12] is a well-known symbolic execution engine that is commonly used as a basis for further research [11, 17, 20, 21, 46, 67]. KLEE interprets LLVM bitcode, the intermediate representation of the LLVM compiler framework. Notably, the C/C++ compiler `clang` can emit LLVM bitcode, which is the IR generation approach proposed originally by KLEE's authors.[1] This makes KLEE unique in our study: it is the only tool that generates IR from source code rather than lifting binaries. It uses the SMT solver STP [10] by default but also supports Z3 [23], which we use as a common ground in our study. KLEE executes all user code at the IR level.

1 We use `clang` version 3.8 with `wllvm` version 1.2.2 to generate LLVM bitcode.

3.3.2 *S2E*

In order to address several perceived shortcomings of KLEE, Chipounov et al. proposed *Selective Symbolic Execution (S2E)* [17]. It builds on top of KLEE but executes programs inside a full virtual operating system. The important difference for our purposes is that S2E generates IR from binaries instead of source code. The program and its environment run inside QEMU [5], a system emulator based on binary translation, and a lifter from QEMU's internal representation to LLVM IR converts the code to a format suitable for consumption by KLEE on demand. Only code interacting with symbolic data is executed symbolically; all other code, including the emulated operating system, runs directly in QEMU. Note that KLEE and S2E use the same symbolic execution engine as well as the same IR but different mechanisms to generate it. This similarity allows us to compare their respective IR generation strategies without the measurement noise from other differences.

3.3.3 *angr*

Shoshitaishvili et al. created angr [74] with the goal of implementing various previously published binary-analysis techniques in a single framework in order to make them comparable. Among many tools for binary analysis, angr provides a symbolic execution engine based on VEX, the intermediate representation used by the Valgrind tools [56]. The system translates binaries to VEX IR, which is then interpreted by angr's symbolic executor. The user can configure whether code that handles only concrete data is passed on to the Unicorn CPU emulator [66]. The symbolic executor is implemented in Python in order to facilitate quick experimentation and scripting. This decision influences execution speed in comparison with tools that are written in lower-level programming languages. We discuss the aspect in more detail during our analysis.[2]

3.3.4 *QSYM*

Yun et al. argue that IR generation and semantic discrepancy between the machine code and IR instruction sets are a major hindrance in modern symbolic execution [86]. To address this problem, they propose QSYM, a symbolic execution engine that directly executes instrumented machine code. The implementation of the symbolic executor is more involved than in conventional IR-based systems, having to handle the large and complex instruction sets of modern CPUs, but the authors argue that the significant performance gains justify the

2 The authors recommend executing angr in `PyPy`, a JIT-compiling implementation of Python, for performance reasons; we use `PyPy` version 5.1.2.

additional implementation work. In our study, we are interested in QSYM precisely because of its lack of IR generation mechanism. The system supplies an interesting data point for our analysis of execution speed and SMT query complexity. QSYM decides at the instruction level whether to execute symbolically or natively.

3.4 EVALUATION

This section conducts the actual measurements. Recall that our ultimate goal is to answer the following research questions:

1. What is the impact on symbolic execution of generating IR from source code as opposed to IR generation from binaries?
2. Does one IR perform better than others when IR generation is comparable? What is the impact of not using IR at all?

In order to answer those high-level questions, we need to decide on concretely measurable properties that supply the necessary evidence. What do we expect of an ideal IR generation technique for symbolic execution? Since we are going to execute the IR, we want it to be easy to interpret efficiently, and we want it to be concise. Moreover, since SMT solving consumes a considerable portion of the overall analysis time, we would like the IR to lead to SMT queries that the solver can answer quickly. We therefore evaluate the various IR generation mechanisms under three aspects:

1. How much does the translation to IR increase or decrease the number of instructions?
2. How efficiently can we execute the resulting IR?
3. How hard are the solver queries derived from the IR?

We first discuss our methodology and the non-trivial task of generating a set of programs that are supported by all the symbolic execution engines we selected. Then we investigate the effect on the number of instructions before presenting the results on execution speed and query complexity. Interested readers will find additional visualizations and a link to raw data in the appendix. We discuss the implications of our results in the next section, where we also answer the research questions.

3.4.1 *Experimental setup*

A core challenge in assessing the impact of IR and IR generation on symbolic execution is that different symbolic execution engines generally differ in many factors, not just the IR generation process. For instance, KLEE and angr differ in how they generate IR, but in

addition to this aspect relevant to our study there are other differences that introduce noise into our measurements:

- One is implemented in C++, the other in Python. We found that this has a major impact on the speed of symbolic execution.
- Their respective execution engines vary in search strategy, i.e., they use different heuristics for prioritizing execution states. Some simple heuristics like depth-first search are supported by both but generally do not lead to interesting paths through the software under test [15, 70].
- The systems have been developed with different goals in mind. While the one focuses on speed and fully automatic execution, the other places some emphasis on scriptability and interactive exploration.
- Implementations of symbolic execution may be faulty. While our goal is to evaluate a given *approach*, we can only analyze the *implementation* at hand and have to trust that it faithfully represents the approach. Previous work has shown that there can be discrepancies [68].

It is therefore difficult to isolate the effects of IR generation from the influence of other differences. One option would be to implement a grand unified symbolic execution engine working on top of the various IRs in order to eliminate most variables. However, the symbolic execution engines we analyze are tuned to the properties of their respective IRs; for instance, KLEE can run optimizations on the input LLVM bitcode that are meant to compensate some shortcomings of the IR generation process and make the IR more suitable for symbolic execution. We felt that running the IR of the various systems in a more generic execution engine would lead to a less fair comparison. Instead, we strove to eliminate as many sources of bias as possible by identifying design decisions that could introduce noise into our measurements (see Section 3.2) and making minimal changes to all systems in order to remove any such differences (discussed below). We believe that such an analysis, despite its possible limitations, yields the most valuable insights into the problem at hand.

While measuring the impact on code size of each system is relatively easy, in order to evaluate the execution speed and the complexity of generated queries we first had to find a set of target programs that all four symbolic execution engines under analysis are able to execute. We remark that this has turned out to be a significant challenge: while the four systems share an overall goal, the specifics vary enough to make it difficult to find programs for which each tool is usable. We discuss the implications for our benchmarks in more detail below.

After some experimentation, we decided to use the programs from DARPA's *Cyber Grand Challenge (CGC)* for our evaluation, mainly for

two reasons: First, the CGC programs have explicitly been designed as a test suite for automated vulnerability detection and exploitation systems. They are supposed to exhibit common code patterns. Moreover, they run on DECREE, a Linux-based operating system with a simplified system call interface, originally designed in order to reduce the engineering burden on the participants in the CGC competition. This makes it easier for us to add missing support to symbolic execution engines. Second, S2E and angr were used by teams participating in the CGC. Therefore, those tools are known to work with the CGC programs. Furthermore, the authors of QSYM evaluate their system on a variant of the CGC binaries in the original publication [86]. The CGC suite contains a total of 131 different programs.

As discussed in Section 3.2.1, the choice of path selection and scheduling algorithms has a major impact on symbolic execution. We eliminate this potential source of noise in our measurements by evaluating concolic execution, i.e., we make symbolic execution follow the path determined by a fixed input. In particular, we use the proofs of vulnerability (PoVs) provided by DARPA for each CGC application. They represent interactions with the applications that exercise bugs. Where multiple such PoVs are available, we choose the first. Our input selection procedure is thus analogous to the QSYM authors' strategy. The motivation to use the PoVs for test input, as outlined by Yun et al. [86], is the assumption that inputs reaching the bugs in the CGC applications exercise interesting portions of code.

DARPA provides the PoVs in a custom XML format designed to describe the interaction with a target application. We wrote a tool that translates the XML description to raw data; we skip applications where the translation of the corresponding PoVs was not possible. This happened in a few cases where the input exercising the vulnerability in a program depended on previous output received from the program—the XML format provides facilities for handling such scenarios, but the same logic cannot be reflected in raw data inputs. We confirmed with the authors of QSYM that this aspect of our procedure is analogous to their evaluation, helping comparability.

All four symbolic execution engines required modifications or extensions for our experiments. We strove to keep our changes to the engines minimal in order to avoid interference, and we have made our modifications available to the community. Concretely, we added 134 lines of code (LoC) to KLEE (partial support for `mmap` and `munmap`), 67 LoC to S2E (time measurements and early termination of execution states), 19 LoC to angr (timing and query logging) and 26 LoC to QSYM (logging as well). We wrote considerably more code, but it is concerned with the generation of suitably compiled programs, conversion of the inputs provided by DARPA into the right formats, proper invocation of the tools, automated measurements, etc.—it does not affect the inner workings of the engines under analysis.

We execute each symbolic execution engine on each CGC application with a timeout of 30 minutes and a memory limit of 24 GB. The experiments run under Ubuntu 16.04 and use one core of an Intel Xeon Gold 6130 CPU each. We skip any applications that are not supported by all engines. Note that, while 30 minutes of symbolic execution would be far too short for vulnerability discovery, we do not let the systems explore the target applications freely. Instead, we execute symbolically along a predetermined path (which, coincidentally, is known to lead to a vulnerability), observing run-time aspects such as the speed of execution and generated SMT queries on the way. The time frame of 30 minutes is sufficient to finish execution in most cases; we exclude any experiments that run into a timeout or exceed the memory quota.

3.4.2 *Benchmark size*

Out of the 131 CGC programs, only 24 execute successfully in all four symbolic execution engines (see Table 3.2). While IR generation is not typically a problem since all systems use mature generators, incompatibilities of the IR execution engines precluded successful analysis in many cases. For example, KLEE immediately exits if the program under test contains floating-point instructions; we compiled the target programs statically to make sure that the offending instructions only occur in programs where they are strictly required, but even so KLEE exhibits the smallest number of supported programs. In the case of angr, its focus on scripting and interactive exploration often renders it too slow to work on large binaries. S2E, in turn, has only recently gained the ability to track data through MMX/SSE registers; in earlier versions, the contents of such registers were concretized, causing the symbolic execution engine to lose track of the corresponding symbolic expressions. Note that SSE registers are used in prominent places, such as the `strcmp` and `strncmp` functions in *GNU libc*. The example of S2E also demonstrates that adding all missing features ourselves was not an option: the code for MMX/SSE register support alone amounts to roughly 1400 lines[3] of C/C++ across various libraries; and this addresses a single limitation in a single tool. In general, the missing features typically require time-consuming engineering—which is presumably why they have not been implemented in the first place. Similar problems have been described by Qu and Robinson [64] and by Xu et al. [85]. Finally, fairness requires us to base our comparison only on targets that are supported by *all* engines, which further restricts the test set.

Table 3.3 provides an overview of the CGC programs that we used for our experiments. When programs can be used successfully for the speed measurements in Section 3.4.4 but not for the assessment of

3 https://github.com/S2E/s2e-env/issues/144

	QSYM	S2E	angr	KLEE	all
Execution speed (Section 3.4.4)	70.2%	66.4%	75.6%	35.1%	18.3%
Query complexity (Section 3.4.5)	57.3%	74.8%	87.8%	38.9%	17.6%

Table 3.2: Percentage of CGC programs (out of 131) that we were able to use per experiment and symbolic execution engine.

query complexity in Section 3.4.5, the reason is often that the generated queries are so complex that the solver times out. In the inverse case, i.e., programs used to measure query complexity but not for execution speed, we encountered a few different cases: angr issues SMT queries for each input byte in every execution, independently of whether the data is used. In some cases, it never encounters instructions that operate on the symbolic input data, so that we do not include the program in the evaluation of execution speed; however, due to the behavior mentioned above, there are still queries whose complexity can be assessed. Moreover, we found S2E's statistical counters to be lagging behind in some cases. In programs with very few symbolic operations, the counters may report zero, resulting in those programs being excluded from the speed measurements. Since we still see SMT queries, however, we include them in the experiments on query complexity.

The lack of extensive tool support is the main reason why we believe it is not currently possible to compare symbolic execution engines on large sets of applications, especially on applications with high complexity. Even assembling a set of 24 applications that work with all four symbolic execution tools in our analysis has cost us significant time and effort. Under such circumstances, is there even value in the comparison? We strongly believe that there is, for two reasons:

1. Even on a limited data set we can see trends; such observations add rigor to a discussion that has until now been driven by intuition and anecdotal evidence.

2. As a community, we should incentivize comparable research—if a new tool in the field cannot meaningfully be compared to existing approaches, we cannot assess its value. We should therefore strive to establish a shared benchmarking methodology and data set; this study attempts to take a step in that direction.

3.4.3 *Code size*

We have previously mentioned the intuition that IR derived from source code contains "more high-level information" than binary-based IR; a more precise way of expressing this intuition is to say that we expect source-derived IR to contain more semantic information per

Name	Size (LoC)	Used in Section		Description
		3.4.4	3.4.5	
CROMU_00020	414	✓	✓	Echo service
CROMU_00043	950		✓	Protocol-aware packet analyzer
KPRCA_00010	1,391	✓	✓	Visualizer for uncompressed PCM audio files in both the time domain and the frequency domain (with FFT)
KPRCA_00011	1,497	✓		Simple movie rental service
KPRCA_00014	970		✓	Basic virtual machine
KPRCA_00021	1,896	✓		Parser for a custom JSON-like data format
KPRCA_00022	1,442	✓	✓	Online job application form, modeled after web applications
KPRCA_00023	1,667	✓		Online job application form, modeled after web applications
KPRCA_00028	1,529	✓		Interpreter for a custom list-based programming language
KPRCA_00031	1,927	✓		Chat server with bots
KPRCA_00037	1,538		✓	Extractor of section and symbol information for CGC executables
KPRCA_00038	4,304	✓		Awk clone
KPRCA_00040	1,599	✓		Custom compression algorithm
KPRCA_00042	1,769	✓		Simple movie rental service
KPRCA_00047	101,921		✓	Optical character recognition (OCR) engine
KPRCA_00053	2,387		✓	Blogging site
NRFIN_00001	647	✓	✓	SNMP-like service
NRFIN_00004	706	✓	✓	Chat bots
NRFIN_00007	3,873	✓		Simulation of mixing chemicals
NRFIN_00011	1,351	✓		A client for HTML-like documents
NRFIN_00015	467	✓	✓	Stack-based virtual machine
NRFIN_00018	230	✓	✓	Matrix arithmetic
NRFIN_00021	398	✓		Trading algorithm simulation
NRFIN_00023	1,752		✓	Electronic trading system for matching buyers and sellers
NRFIN_00026	37,288	✓	✓	Packet parser
NRFIN_00029	1,998		✓	UTF-enabled file server
NRFIN_00032	4,053	✓		Network protocol dissector
NRFIN_00035	1,266		✓	PLC simulation
NRFIN_00036	667	✓	✓	Personal finance management tool
NRFIN_00038	2,166	✓	✓	Stateful session-based network service
NRFIN_00040	1,766		✓	Regular language recognition and enumeration
NRFIN_00041	1,446	✓	✓	Marine tracking system fashioned after AIS
NRFIN_00042	968	✓	✓	Memory as a service
YAN01_00011	398		✓	Word completion game
YAN01_00012	270		✓	Stack-based virtual machine

Table 3.3: Details of the CGC programs used in our measurements of execution speed (Section 3.4.4) and query complexity (Section 3.4.5).

IR statement than IR derived from binaries.[4] In order to test this hypothesis we apply the IR generation techniques under analysis to a fixed set of programs and compare the resulting number of IR instructions. The base line for our experiments is the number of machine-code instructions.

In addition to the CGC programs discussed above, we use the programs of version 8.30 of the *coreutils* suite [28] for this comparison; they are a popular benchmark in the literature on symbolic execution. For each binary in the set of test programs (i.e., CGC and coreutils), we recover the CFG with angr and subsequently apply each symbolic execution engine's IR translation mechanism to all discovered basic blocks. This requires wrapping the relevant parts of code in S2E and angr: the former exposes the translation component as a shared library that we can use from a C++ program, whereas the latter offers a Python interface which we use from a custom script. QSYM and KLEE do not require custom extensions for this step of our study: the former works directly on machine code, so that no translation is necessary, and the latter conveniently uses the output of the C/C++ compiler `clang`.

For comparison, we conducted some further experiments on the programs of the coreutils suite:

- We added the results of McSema, a static translator from machine code to LLVM bitcode [24] based on the commercial disassembler IDA Pro. Note that we intentionally used McSema unmodified for best performance, meaning that it employed IDA Pro for disassembly rather than angr. While we had initially hoped to be able to run KLEE on the bitcode that McSema generates, we found that there are incompatibilities in the respective sets of supported bitcode instructions; substantial changes would be required to make the two systems compatible.

- We compiled the coreutils binaries for ARM, using the target `arm-none-eabi`, and ran angr's IR generation on them. The other symbolic execution engines do not support ARM or, in the case of KLEE, the IR does not differ significantly.

We compare the number of generated IR instructions to the corresponding number of machine instructions, resulting in a quantity that we call *inflation factor*. Table 3.4 shows the results of our measurements, and Figure 3.2 visualizes the data.

We see that, in general, the binary-based techniques produce a higher number of IR instructions than KLEE's source-based translation; of course, there are many factors involved in the size of the generated

4 There is the additional effect that some information is actually lost during compilation, such as buffer sizes [19]. This is a concern for security checks that may be part of a symbolic execution engine but does not affect the core components of symbolic execution that we focus on in this study (see Figure 3.1).

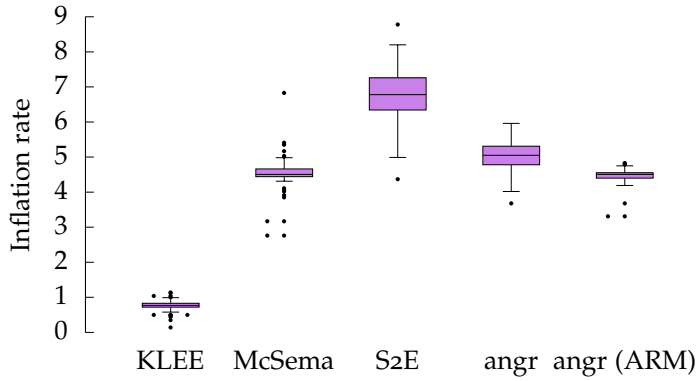

Figure 3.2: Inflation factor per IR generation mechanism, i.e., the number of generated IR instructions per machine-code instruction, across all tested programs (123 CGC and 106 coreutils binaries). The box encloses the second and third quartile of the data with a horizontal line marking the median. The whiskers include data points up to 1.5 times the interquartile range away; outliers beyond that point are depicted individually.

IR generator	IR	CGC	coreutils
QSYM	Machine code	1.00	1.00
KLEE (clang)	LLVM bitcode	0.74	0.78
S2E	LLVM bitcode	6.68	6.29
angr (libvex)	VEX IR	4.57	5.35
McSema	LLVM bitcode		4.54
angr on ARM (libvex)	VEX IR		4.40

Table 3.4: Mean inflation factor per IR generation mechanism and data set, i.e., average number of generated IR instructions per machine-code instruction as shown in Figure 3.2. The CGC data set contains 123 programs, the coreutils suite 106 programs.

translation artifacts. The comparison is most meaningful when the target of the translation process is the same IR, which removes one variable from the analysis. Therefore, the cases of S2E and McSema are of particular interest: both tools start at the binary level and produce LLVM IR, so we can compare their results with the source-based LLVM IR generated by KLEE. Note that, while the IR produced from source code is rather succinct, in almost all cases containing less instructions than the equivalent machine code and reaching an inflation factor below 1 on average, the corresponding IR generated from binaries increases the number of instructions by a factor of above 6 for S2E and 4.54 for McSema. S2E's higher inflation factor may be due to the two IR translations (i.e., machine code to QEMU IR to LLVM IR). Furthermore, it is interesting to see that angr's translation to VEX IR yields an increase in the number of instructions that is similar

to the binary-based tools translating to LLVM IR; in fact, manual analysis suggests that the semantic content of instructions in VEX IR is comparable to LLVM IR. The ARM experiment confirms the overall picture on a different architecture. On average, we find that the IR generated from binaries is considerably larger than source-based IR.

In summary, the data supports the hypothesis that a source-based approach has more high-level information available to generate a succinct IR.

3.4.4 *Execution speed*

The first aspect of symbolic execution that we are interested in is how well the generated IR is suited for execution. There is a spectrum between QSYM, which forgoes translation to IR entirely and directly executes instrumented machine code, computing symbolic constraints on the fly, and KLEE, which interprets high-level IR derived from source code. Intuitively, we would expect IR that is close to (or identical with) machine code to be efficiently executable, while a more abstract representation may be more suited to static analysis but slower in execution.

A major challenge in comparing the execution speed of the four symbolic execution engines is that they use very different strategies on the matter of interleaving concrete and symbolic execution, an issue that was briefly mentioned in Section 3.2.3. In general, code can be executed in one of four modes:

NATIVE The simplest case is native execution of machine code on the CPU, possibly with some sort of instrumentation. This is what QSYM does for concrete execution, and S2E uses QEMU with KVM enabled, resulting in a similar effect.

NATIVE (EMULATED) This case is similar to raw native execution, except that the CPU is emulated. angr uses emulated native execution for code that does not work with symbolic data.

IR (SYMBOLIC) When code works with symbolic data, it has to be translated to IR, which is then interpreted symbolically. All four systems in our study support this mode; in the case of QSYM, the "IR" is machine code.

IR (CONCRETE) KLEE does not support interleaved concrete execution, so even code that works with only concrete data is executed at the IR level. Similarly, angr may heuristically choose to run even concrete computations with IR in situations where the cost of switching back and forth between IR and emulated native execution would otherwise be too high.

Table 3.5 shows the use of the various execution modes by the systems in our analysis. We are interested in the execution of IR, so

	Native	Native (emulated)	IR (concrete)	IR (symbolic)
QSYM	✓			✓
S2E	✓			✓
angr		✓	✓	✓
KLEE			✓	✓

Table 3.5: Execution modes used by the symbolic execution engines in our study. For QSYM, the "IR" in symbolic mode is machine code.

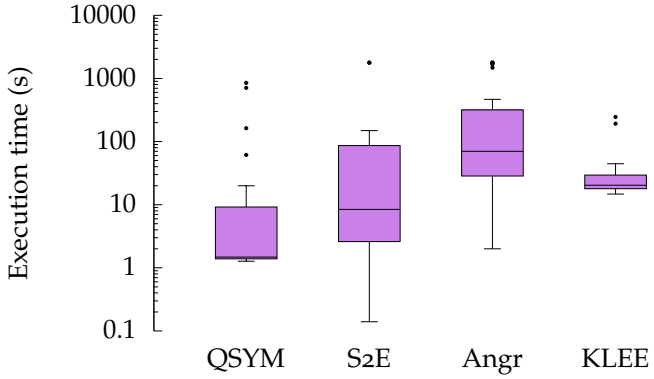

Figure 3.3: Absolute execution times across 24CGC programs.

for the purpose of this study we count only instructions executed in one of the two IR modes, and we only measure the time spent in those modes.

Apart from the difficulty of handling different execution modes, the question of execution speed is particularly prone to being influenced by other factors than merely the IR generation process. In particular, the programming language that a symbolic executor is implemented in has a large effect on how fast it can execute its IR (see Table 3.1). There is little we can do to eliminate this bias (short of reimplementing all systems in a common language); we will take it into account when interpreting our results.

In order to assess the speed of execution we count the number of instructions executed at the IR level and the time spent on said execution while conducting the experiments described in Section 3.4.1. In terms of Table 3.5, we capture the last two columns, which contains any possible execution of IR. Figure 3.3 displays the absolute execution times that we measured; we see that angr consumes an order of magnitude more time than S2E, which is in turn significantly slower than QSYM. Based on our measurements, we compute a quantity that we call *execution rate*; it represents the number of instructions executed per unit of time. Figure 3.4 shows the execution rates in terms of each system's own IR instructions per time. For comparability between different IRs, we translate the execution rates from IR instructions per time to the common basis of machine instructions per time using

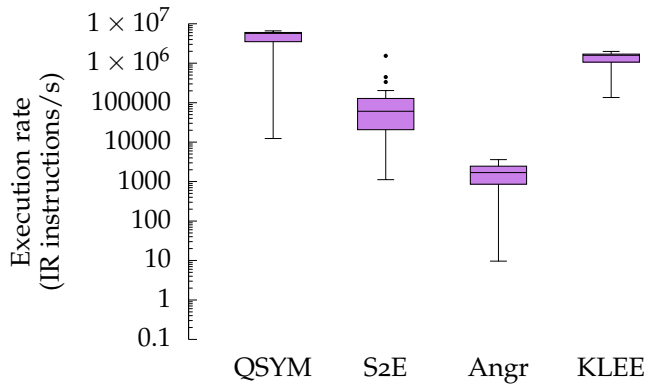

Figure 3.4: Execution speed of symbolically executed instructions across 24 CGC programs. Higher rates mean faster execution.

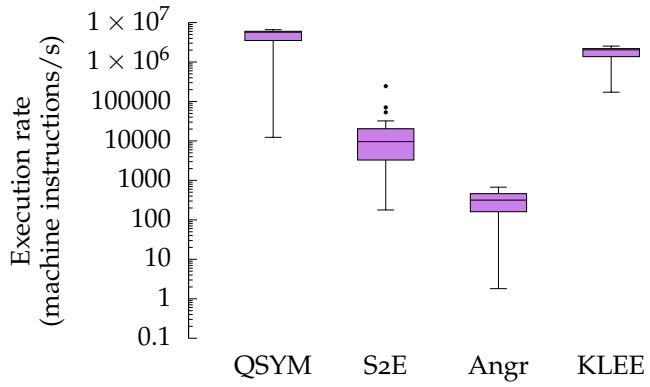

Figure 3.5: Execution speed of symbolically executed instructions, translated to the common basis of machine-code instructions, across 24 CGC programs. Higher rates mean faster execution.

the inflation factors from Table 3.4. Figure 3.5 shows those final results. In other words, we obtain a measure of execution speed that is comparable across different IR generation processes.

We observe that QSYM executes its "IR" the fastest, followed by KLEE, S2E and angr. This matches our intuition, given that QSYM uses the lowest-level IR and implements its symbolic component in C++. KLEE and S2E share a common basis, but while KLEE executes a very concise IR (see Section 3.4.3), S2E has significantly more instructions to interpret. Moreover, S2E has to generate IR on the fly while, in the case of KLEE, IR generation is a preprocessing step. We largely attribute angr's lower execution rate to the fact that its symbolic reasoning is implemented in Python, whereas S2E uses C++.

In summary, the measurement of execution rates supports the hypothesis that low-level IR can be executed faster than high-level IR, and that LLVM bitcode and VEX IR have quite similar properties when it comes to IR interpretation. Note that "low-level IR" refers to the level of abstraction of the IR language, not of the artifact that

the IR was generated from. For instance, raw machine code (QSYM) is executed faster than LLVM bitcode (KLEE and S2E). However, the source of the translation still impacts the concision of the generated IR (see Section 3.4.3)—e.g., LLVM bitcode generated from binaries (S2E) is typically more verbose than bitcode generated from source code (KLEE) and hence requires more time to perform equivalent computations.

3.4.5 *Query complexity*

Along with IR execution, SMT solving is one of the major workloads in symbolic execution [51, 59]. Consequently, there is promise in exploring to which extent the IR generation process impacts the difficulty of the SMT queries arising during execution. Intuitively, if IR carries a lot of semantic information it should be possible for the symbolic executor to formulate succinct queries. For example, consider the program in Listing 3.1; it just reads five bytes from standard input, checks a number of conditions on the input and prints a result message. Listings 3.2 and 3.3 show the queries generated by S2E and KLEE, respectively, for the C expression `data[3] == 55`. While the semantic content is the same in both queries, note how S2E expresses the equality check in bit-wise AND and OR operations as well as a bit-vector addition; the query is more similar to machine code than to the original C code. The KLEE-generated query, in contrast, resembles the source code rather closely. This example illustrates the notion that queries with identical semantics can be formulated in different ways, which may differ in the difficulty they pose for SMT solvers.

Listing 3.1: A simple program to demonstrate SMT queries.

```
#include <stdio.h>

int main(int argc, char* argv[]) {
    char data[5];

    for (int i = 0; i < 5; i++)
        data[i] = getchar();

    if (data[0] > 15 && data[1] == 32 && data[2] > 27 &&
        data[2] < 100 && data[3] == 55 && data[4] == 123)
        printf("Correct!\n");
    else
        printf("Try again...\n");

    return 0;
}
```

Listing 3.2: Part of S2E's assertion for Listing 3.1. We use standard SMT-LIB syntax [4] for SMT queries.

```
(= (_ bv0 64)
   (bvand
    (bvadd
     ;; 0xFFFFFFFFFFFFFFC9
     (_ bv18446744073709551561 64)
     ((_ zero_extend 56)
      ((_ extract 7 0)
       (bvor
        (bvand
         ((_ zero_extend 56) (select stdin (_ bv3 32)))
         ;; 0x00000000000000FF
         (_ bv255 64))
        ;; 0xFFFF88000AFDC000
        (_ bv18446612132498620416 64)))))
    (_ bv255 64)))
```

Listing 3.3: Part of KLEE's assertion for Listing 3.1.

```
(= (_ bv55 8)
   ((_ extract 7 0)
    ((_ zero_extend 24) (select stdin (_ bv3 32)))))
```

In general, assessing the difficulty of SMT queries is not an easy task. Even with a proper definition of the elusive concept of "difficulty", there may be no effective means of measuring it. We observe that, from a practical point of view, the essential property of an "easy" query is that the solver can answer it fast. Therefore, our approach is to run all symbolic execution engines on the same fixed paths in concolic mode and record the queries that are sent to the solver. We then run the solver on those queries in isolation and measure its response time. This allows us to assess the average solver effort for each tool on identical workloads, isolated from external factors like IR execution speed.

We measure the time taken by Z3 to solve all the logged queries of successful executions as per Section 3.4.1. The four symbolic execution engines install different versions of Z3 (see Table 3.1); for comparability, we picked one and used it for all measurements. We chose S2E's build of Z3 because it is the most recent among the four, so we expect it to gracefully handle the queries generated by the other engines. Note that we deliberately do not set a timeout for individual queries: We are interested in how long a query would run to completion—i.e., its *complexity*—instead of just the time that it would be allowed to run in practice.

Figure 3.6 visualizes the absolute number of queries generated by each system. We note that angr and KLEE tend to issue more queries than S2E and QSYM. However, we attribute the differences to the

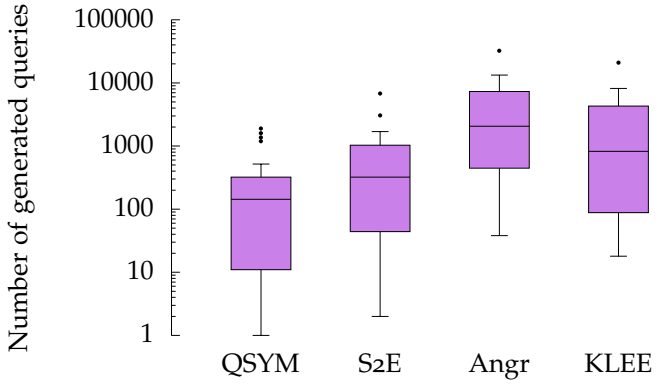

Figure 3.6: Absolute number of queries generated by the symbolic execution engines across 23 CGC programs.

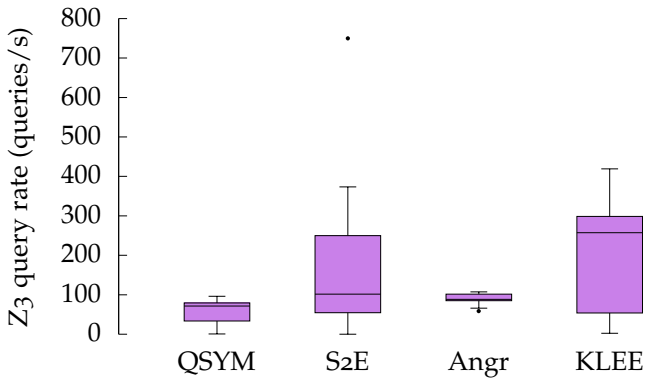

Figure 3.7: Comparison of the query rates for each system (using a common solver) as a proxy for query complexity, across 23 CGC programs. Higher rates indicate queries that are easier to solve. Note the differences in median.

varying degrees of instrumentation in the implementations rather than the IR or its generation. For instance, KLEE performs bounds checks on every memory access and tests whether pointers may be null; QSYM only involves the solver when the control flow depends on symbolic data and defers any security checks to the fuzzer that is expected to run concurrently (see Figure 3.1).

Figure 3.7 shows the resulting *query rates*, i.e., the number of queries that Z3 can solve in a fixed amount of time. We see that angr and QSYM exhibit lower query rates than KLEE, whose median rate is significantly higher. S2E's queries fall into a range similar to KLEE's (which is sensible because S2E is based on KLEE), but note that S2E's median is considerably lower and more in line with angr and QSYM. In general, it seems that the three binary-based symbolic execution systems generate more difficult queries than the source-based system KLEE. Moreover, the observation that both KLEE and S2E issue relatively easy queries in many cases supports the notion

that LLVM IR is beneficial for deriving SMT queries. However, we cannot rule out the possibility of KLEE generating simpler queries than the other systems due to implementation details; since S2E is based on KLEE, it would inherit the same advantage.

3.5 DISCUSSION

In this section, we interpret the results of our evaluation and discuss their significance.

3.5.1 *Results*

We have measured the impact of IR generation on code size, the suitability of different IRs for symbolic execution, and the complexity of the resulting SMT queries. In summary, we have found the following:

- For code size, the most important factor is whether IR is generated from source code or binaries. While source-based IR is often more succinct than machine code, binary-based IR tends to inflate the code by a factor between 3 and 7.
- We do not observe a significant difference in execution speed between LLVM bitcode and VEX IR that could not be attributed to implementation aspects. QSYM, however, gains a distinct advantage in speed by dispensing with a traditional IR and instrumenting machine code directly, at the expense of portability.
- When generated from machine code, LLVM bitcode and VEX IR lead to queries of similar complexity; queries derived directly from machine code are in the same range. S2E does generate simpler queries than angr and QSYM in some cases, but the median query rate is similar. Source-based IR, however, appears to reliably lead to simpler queries during symbolic execution.

Therefore, we are now in a position to answer our original research questions.

When source is available, should we generate IR from source code or binaries? We discovered that query complexity is lower when IR is generated from source code. Of course, we acknowledge that source code is not always available and that sometimes low-level information is exactly what one is interested in; therefore, there are good reasons for binary-based symbolic execution as well.

Does any IR perform better than others? We found that the level of abstraction of the IR is important for execution speed; in particular, executing machine code directly yields performance benefits. When comparing the "traditional" IRs, there is no observable difference between LLVM bitcode (generated from binaries) and VEX IR in our measurements; we believe that, for choosing one or the other, practical

concerns such as API stability and the availability of language bindings are more important factors than the impact on symbolic execution.

To summarize, we show that the most important influence on query complexity is whether the IR is generated from source code or binaries, whereas execution speed is mostly affected by the level of abstraction of the IR, with raw machine code performing best. This creates an interesting tension in the design of symbolic execution engines: for highest execution speed, execution should be based on low-level instructions, whereas the best solver performance is achieved with queries generated from high-level code. We address this challenge in Chapter 4.

3.5.2 *Future work*

Our study focuses on the *speed* of symbolic execution, and we argue that faster execution and SMT solving yield more exploration in the same time, thus increasing the probability of discovering vulnerabilities. An interesting direction for future work, especially in the context of combined fuzzing and symbolic execution, would be to assess the *quality* of new program inputs generated by symbolic execution. A measurable notion of quality should include factors like the resulting increase in code coverage, similarity to existing test cases (for easier bug triage), redundancy of test inputs, and "directedness" towards interesting pieces of code, among others. After finding a quantifiable definition of test case quality, one would have to develop a sound methodology to actually measure it; we believe that the results could be very interesting for the community.

In a similar vein, it would be interesting to evaluate what makes queries hard for a solver. We showed in our study that IR generated from binaries leads to harder SMT queries than IR generated from source code—what is the root cause of the difference in difficulty? Compiler optimizations come to mind as a possible source of complexity. However, we expect at least some of them to simplify reasoning about code rather than making it harder: for instance, when a multiplication is replaced with a bit shift during strength reduction, the optimization should not only speed up the program but also reduce the difficulty of the corresponding queries. A systematic evaluation of the sources of complexity in the queries that arise during symbolic execution might lead to IR generators that produce more "solver-friendly" IR.

Finally, in our study we have analyzed the impact of IR and IR generation on specific aspects of symbolic execution, but we have not evaluated the effect on the overall goal: how does the IR aspect impact bug discovery? While this is a highly interesting question, we believe that answering it is a hard challenge. The different symbolic execution engines use vastly different strategies to generate new test cases,

involving different choices in the selection and configuration of the SMT solver, the caching and preprocessing of queries, the soundness requirements on the analysis, etc. Figuratively speaking, all the components of symbolic execution depicted in Figure 3.1 would introduce bias in such an end-to-end comparison. We would be delighted to see more modularization in this space: if the individual components of symbolic execution engines were interchangeable, measuring the impact of a single choice on the overall goal would become much easier.

3.5.3 *Limitations*

Comparing design decisions of symbolic execution engines in isolation is a complicated matter: we have discussed numerous ways for seemingly unrelated design decisions to threaten the accuracy of our measurements. And while we have invested significant effort to eliminate such noise from our experiments, there may be effects that we could not fully remove. Moreover, some differences cannot be reasonably eliminated, such as the impact of the respective programming languages that the systems are built in. Finally, we have run our experiments on a limited set of test programs that may not be representative. We would like to explicitly encourage follow-up work that strives to identify remaining biases in the comparison of symbolic execution engines.

3.5.4 *Remark: programming languages*

We note in passing that the choice of programming language plays an important role in positioning a symbolic execution engine. For example, KLEE is written in C++, which gives it considerable performance advantages over angr, implemented in Python. However, we know from experience that modifying the former is much more time-consuming than building on top of the latter; we attribute the difference to the different characteristics of the respective programming languages. There is, of course, no perfect solution; the ideal choice for a given project will vary depending on, among a lot of other factors, whether production use or experimentation and exploration are the main goal. However, we think it is important to consider such aspects upfront and to make a conscious decision.

3.6 RELATED WORK

To the best of our knowledge, the impact of the choice of IR and IR generation process on symbolic execution has not been studied before. However, our work builds on top of various previous results. In this section, we frame our study in the context of the current state of

the art, focusing in particular on symbolic execution and intermediate representations for static and dynamic analysis.

3.6.1 *Symbolic execution*

Symbolic execution lies on a spectrum between more rigorous approaches, such as model checking [26, 65], and techniques that sacrifice soundness for practicality, such as fuzz testing [25]. Apart from the four symbolic execution engines that form the basis of our analysis, namely KLEE [12], S2E [17], angr [74] and QSYM [86], each representing a design category as described in Section 3.3, several others have been proposed and implemented. Manticore [81] is similar in focus to angr and implemented in Python as well but does not use any intermediate representation. Triton [69] is based on dynamic binary translation, like QSYM. Mayhem [15], based on BAP [9], is the winner of the DARPA CGC competition (but not freely available, and BAP alone does not support symbolic execution in recent versions). SAGE [33] is a closed-source system developed by Microsoft, following a concolic execution approach. Inception [21], based on KLEE, is among the few symbolic execution engines with support for ARM, and it addresses the challenge of handling inline assembly in source-based symbolic execution. However, it targets microcontrollers that run their target software directly, without an operating system. This difference in focus renders it hard to compare to the four systems in our study. Finally, various other systems extend KLEE with additional functionality, e.g., localized vulnerability detection [67], support for floating-point arithmetic [20], parallel analysis [11], or state merging [46]. Recently, combining symbolic execution with fuzzing has been shown to hold great promise [77, 86].

Our study focuses on a particular aspect in the design and implementation of symbolic execution systems. In a similar spirit, previous work has focused on the choice of SMT solvers [59] and the impact of incremental SMT solving [51]. Kapus and Cadar check the correctness of symbolic execution engines via differential testing [39] (whereas we focus on performance). Baldoni et al. [3] cover the general subject area of symbolic execution, and Xu et al. survey challenges of the field [84].

3.6.2 *Intermediate representations*

There are a variety of intermediate representations. LLVM bitcode [49], employed by KLEE and S2E, was originally designed for use inside compilers. VEX [56], used by angr, targets binary instrumentation and was conceived for the Valgrind framework. Others, such as REIL [50] and BIL [9] have been developed specifically for security analysis. Kim et al. [42] investigate the semantic correctness of lifters for many intermediate representations. Their work is orthogonal to ours: we

assess the impact of the IR and the associated generation process on symbolic execution (presupposing correctness), while they focus on the semantic correctness of the IR generators.

3.7 CONCLUSION

We have presented a framework for comparing different symbolic execution engines and applied it to the question of how IR and IR generation impact symbolic execution. We believe that such systematic evaluation forms a much better basis for design decisions than anecdotal evidence or common belief. It is our hope that this study lays the groundwork for further comparison of specific design aspects in symbolic execution, ultimately leading to more principled decisions and, hopefully, more efficient systems.

AVAILABILITY

We have made all code and data used in this study available to the community at http://www.s3.eurecom.fr/tools/symbolic_execution/ir_study.html, hoping that it will benefit future research.

4 COMPILER-BASED SYMBOLIC EXECUTION

A major impediment to practical symbolic execution is speed, especially when compared to near-native speed solutions like fuzz testing. Based on the insights gained in the previous chapter, we propose a compilation-based approach to symbolic execution that performs better than state-of-the-art implementations by orders of magnitude. We present SymCC, an LLVM-based C and C++ compiler that builds concolic execution right into the binary. It can be used by software developers as a drop-in replacement for `clang` and `clang++`, and we show how to add support for other languages with little effort. In comparison with KLEE, SymCC is faster by up to three orders of magnitude and an average factor of 12. It also outperforms QSYM, a system that recently showed great performance improvements over other implementations, by up to two orders of magnitude and an average factor of 10. Using it on real-world software, we found that our approach consistently achieves higher coverage, and we discovered two vulnerabilities in the heavily tested OpenJPEG project, which have been confirmed by the project maintainers and assigned CVE identifiers.

4.1 INTRODUCTION

Despite the increase in popularity of symbolic execution, performance has remained a core challenge for symbolic execution. Slow processing means less code executed and tested per time, and therefore fewer bugs detected per invested resources. Several challenges are commonly identified, one of which is slow code execution: Yun et al. have recently provided extensive evidence that the execution component is a major bottleneck in modern implementations of symbolic execution [86]. We propose an alternative execution method and show that it leads to considerably faster symbolic execution and ultimately to better program coverage and more bugs discovered.

Let us first examine how state-of-the-art symbolic execution is implemented. With some notable exceptions, most implementations translate the program under test to an intermediate representation (e.g., LLVM bitcode), which is then executed symbolically (see Section 2.2). Conceptually, the system loops through the instructions of the target program one by one, performs the requested computations and also keeps track of the semantics in terms of any symbolic input. This is essentially an interpreter! More specifically, it is an interpreter for

the respective intermediate representation that traces computations symbolically in addition to the usual execution.

Interpretation is, in general, less efficient than compilation because it performs work at each execution that a compiler has to do only a single time [37, 83]. Our core idea is thus to apply "compilation instead of interpretation" to symbolic execution in order to achieve better performance. But what does compilation mean in the context of symbolic execution? In programming languages, it is the process of replacing instructions of the source language with sequences of machine code that perform equivalent actions. So, in order to apply the same idea to symbolic execution, we *embed* the symbolic processing into the target program. The end result is a binary that executes without the need for an external interpreter; it performs the same actions as the target program but additionally keeps track of symbolic expressions. This technique enables it to perform any symbolic reasoning that is conventionally applied by the interpreter, while retaining the speed of a compiled program.

Interestingly, a similar approach was used in early implementations of symbolic execution: DART [32], CUTE [71] and EXE [13] instrument the program under test at the level of C source code. In comparison with our approach, however, they suffer from two essential problems:

1. Source-code instrumentation ties them to a single programming language. Our approach, in contrast, works on the compiler's intermediate representation and is therefore independent of the source language.

2. The requirement to handle a full programming language makes the implementation very complex [32]; the approach may be viable for C but is likely to fail for larger languages like C++. Our compiler-based technique only has to handle the compiler's intermediate representation, which is a significantly smaller language.

The differences are discussed in more detail in Section 4.6.

We present an implementation of our idea, called SymCC, on top of the LLVM framework. It takes the unmodified LLVM bitcode of a program under test and compiles symbolic execution capabilities right into the binary. At each branch point in the program, the "symbolized" binary will generate an input that deviates from the current execution path. In other words, SymCC produces binaries that perform concolic execution, a flavor of symbolic execution that does not follow multiple execution paths at the same time but instead relies on an external entity (such as a fuzzer) to prioritize test cases and orchestrate execution (see Section 2.1 for details).

In the most common case, SymCC replaces the normal compiler and compiles the C or C++ source code of the program under test

into an instrumented binary.[1] As such, SymCC is designed to analyze programs for which the source code (or at least LLVM bitcode) is available, for example during development as part of the secure development life cycle. It can, however, handle binary-only libraries and inline assembly gracefully. We discuss this aspect in more detail in Section 4.5.

Figure 4.2 shows an example interaction with SymCC: We first compile the program displayed in Listing 4.1, simulating a log-in interface. Then we run the program with an initial test input and demonstrate that concolic execution generates a new test input allowing us to access the most interesting portion of the program. While this is a very basic example, we hope that it gives the reader an idea of how SymCC can be used.

Listing 4.1: A sample C++ program that emulates a log-in interface. The most interesting portion of the program is reached when the user inputs "root".

```
#include <iostream>

int main(int argc, char *argv[]) {
  std::cout << "What's your name?" << std::endl;
  std::string name;
  std::cin >> name;

  if (name == "root")
    std::cout << "What is your command?" << std::endl;
  else
    std::cout << "Hello, " << name << "!" << std::endl;

  return 0;
}
```

Listing 4.2: A shell session that demonstrates how a user would compile and run the program from Listing 4.1 with SymCC. Lines prefixed with a dollar sign indicate commands entered by the user. Note how the analysis proposes "root" as a new test input.

```
$ sym++ -o login_symcc login.cpp
$ export SYMCC_OUTPUT_DIR=/tmp/symcc
$ echo "john" | ./login_symcc 2>/dev/null
What's your name?
Hello, john!
$ cat /tmp/symcc/000008-optimistic
root
```

In larger software projects, it is typically sufficient to export `CC=symcc` and `CXX=sym++` before invoking the respective build system; it will pick

1 Support for additional source languages can be added with little effort; see Section 4.3.6.

up the compiler settings and build an instrumented target program transparently.

Note that SymCC fulfills the requirements for fast symbolic execution that we found in the previous chapter (see, in particular, Section 3.5.1): execution is based on native code, yielding high execution speed, while queries are derived from the compiler's higher-level program representation, which leads to simpler SMT queries. Moreover, the decision to make SymCC a drop-in replacement for the regular compiler is a direct consequence of the usability challenges we encountered with other symbolic execution engines (see Section 3.4.1).

In summary, this chapter makes the following contributions:

1. We propose compilation-based symbolic execution, a technique that provides significantly higher performance than current approaches while maintaining low complexity.
2. We present SymCC, our open-source implementation on top of the LLVM framework.
3. We evaluate SymCC against state-of-the-art symbolic execution engines and show that it provides benefits in the analysis of real-world software, leading to the discovery of two critical vulnerabilities in OpenJPEG.

SymCC is publicly available at `http://www.s3.eurecom.fr/tools/symbolic_execution/symcc.html`, where we also provide the raw results of our experiments, as well as the tested programs.

4.2 COMPILATION-BASED SYMBOLIC EXECUTION

We now describe our compilation-based approach, which differs from both conventional IR-based and IR-less symbolic execution but combines many of their advantages. The high-level goal of our approach is to accelerate the *execution* part of symbolic execution (as outlined in Section 2.1) by compiling symbolic handling of computations into the target program. The rest of this section is devoted to making this statement more precise; in the next section, we describe the actual implementation.

4.2.1 *Overview*

An interpreter processes a target program instruction by instruction, dispatching on each opcode and performing the required actions. A compiler, in contrast, passes over the target ahead of time and replaces each high-level instruction with a sequence of equivalent machine-code instructions. At execution time, the CPU can therefore run the program directly. This means that an interpreter performs work during every execution that a compiler needs to do only once.

In the context of symbolic execution, current approaches either interpret (in the case of IR-based implementations) or run directly on the CPU but with an attached observer (in IR-less implementations), performing intermittent computations that are not part of the target program. Informally speaking, IR-based approaches are easy to implement and maintain but rather slow, while IR-less techniques reach a high performance but are complex to implement. A core claim of the thesis is that we can combine the advantages of both worlds, i.e., build a system that is easy to implement yet fast; this chapter makes a first step in that direction. In essence, we compile the logic of the symbolic interpreter (or observer) into the target program. Contrary to early implementations of symbolic execution [13, 32, 71], we do not perform the embedding at the source-code level but instead work with the compiler's intermediate representation, which allows us to remain independent of the source language that the program under test is written in, as well as independent of the target architecture (cf. Section 4.6).

Listing 4.3: An example function in LLVM bitcode. It takes two integers and checks whether the first is exactly twice the second.

```
1 define i32 @is_double(i32, i32) {
2   %3 = shl nsw i32 %1, 1
3   %4 = icmp eq i32 %3, %0
4   %5 = zext i1 %4 to i32
5   ret i32 %5
6 }
```

To get an intuition for the process, consider the example function in Listing 4.3. It takes two integers and returns 1 if the first integer equals the double of the second, and 0 otherwise. How would we expect compiler-based symbolic execution to transform the program in order to capture this computation symbolically? Listing 4.4 shows a possible result. The inserted code calls out to the run-time support library, loaded in the same process, which creates symbolic expressions and eventually passes them to the symbolic backend in order to generate new program inputs (not shown in the example). Note that the transformation inserting those calls happens at compile time; at run time, the program "knows" how to inform the symbolic backend about its computations without requiring any external help and thus without incurring a significant slowdown. Figure 4.1 summarizes the approach; note how it contrasts with the conventional techniques depicted in Figures 2.2 and 2.3. We will now go over the details of the technique.

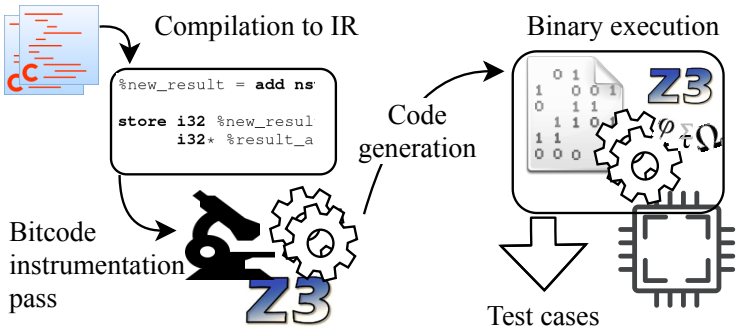

Figure 4.1: Our compilation-based approach compiles symbolic execution capabilities directly into the target program.

Listing 4.4: Simplified instrumentation of Listing 4.3. The called functions are part of the support library. The actual instrumentation is slightly more complex because it accounts for the possibility of non-symbolic function parameters, in which case the symbolic computation can be skipped.

```
1  define i32 @is_double(i32, i32) {
2    ; symbolic computation
3    %3 = call i8* @_sym_get_parameter_expression(i8 0)
4    %4 = call i8* @_sym_get_parameter_expression(i8 1)
5    %5 = call i8* @_sym_build_integer(i64 1)
6    %6 = call i8* @_sym_build_shift_left(i8* %4, i8* %5)
7    %7 = call i8* @_sym_build_equal(i8* %6, i8* %3)
8    %8 = call i8* @_sym_build_bool_to_bits(i8* %7)
9
10   ; concrete computation (as before)
11   %9 = shl nsw i32 %1, 1
12   %10 = icmp eq i32 %9, %0
13   %11 = zext i1 %10 to i32
14
15   call void @_sym_set_return_expression(i8* %8)
16   ret i32 %11
17 }
```

4.2.2 *Support library*

Since we compile symbolic execution capabilities into the target program, all components of a typical symbolic execution engine need to be available. We therefore bundle the symbolic backend into a library that is used by the target program. The library exposes entry points into the symbolic backend to be called from the instrumented target, e.g., functions to build symbolic expressions and to inform the backend about conditional jumps.

4.2.3 *Symbolic handlers*

The core of our compile-time transformation is the insertion of calls to handle symbolic computations. The compiler walks over the entire program and inserts calls to the symbolic backend for each computation. For example, where the target program checks the contents of two variables for equality, the compiler inserts code to obtain symbolic expressions for both operands, to build the resulting "equals" expression and to associate it with the variable receiving the result (see expression `%7` in Listing 4.4). The code is generated at compile time and embedded into the binary. This process replaces a lot of the symbolic handling that conventional symbolic execution engines have to perform at run time. Our compiler instruments the target program exactly once—afterwards, the resulting binary can run on different inputs without the need to repeat the instrumentation process, which is particularly effective when combined with a fuzzer. Moreover, the inserted handling becomes an integral part of the target program, so it is subject to the usual CPU optimizations like caching and branch prediction.

4.2.4 *Concreteness checks*

It is important to realize that each inserted call to the run-time support library introduces overhead: it ultimately invokes the symbolic backend and may put load on the SMT solver. However, involving the symbolic backend is only necessary when a computation receives symbolic inputs. There is no need to inform the backend of fully concrete computations—we would only incur unnecessary overhead (as discussed in Section 2.2.3). There are two stages in our compilation-based concolic approach where data can be identified as concrete:

COMPILE TIME Compile-time constants, such as offsets into data structures, magic constants, or default return values, can never become symbolic at run time.

RUN TIME In many cases, however, the compiler cannot know whether data will be concrete or symbolic at run time, e.g., when it is read from memory: a memory cell may contain either symbolic or concrete data, and its concreteness can change during the course of execution. In those cases, we can only check at run time and prevent invocation of the symbolic backend dynamically if all inputs of a computation are concrete.

Consequently, in the code we generate, we omit calls to the symbolic backend if data is known to be constant at compile time. Moreover, in the remaining cases, we insert run-time checks to limit backend calls to situations where at least one input of a computation is symbolic (and thus the result may be, too).

4.3 IMPLEMENTATION OF SYMCC

We now describe SymCC, our implementation of compiler-based symbolic execution. We built SymCC on top of the LLVM compiler framework [49]. Compile-time instrumentation is achieved by means of a custom compiler pass, written from scratch. It walks the LLVM bitcode produced by the compiler frontend and inserts the code for symbolic handling (as discussed in Section 4.2.3). The inserted code calls the functions exported by the symbolic backend: we provide a thin wrapper around the Z3 SMT solver [23], as well as optional integration with the more sophisticated backend of QSYM [86]. The compiler pass consists of roughly 1,000 lines of C++ code; the run-time support library, also written in C++, comprises another 1,000 lines (excluding Z3 and the optional QSYM code). The relatively small code base shows that the approach is conceptually simple, thus decreasing the probability of implementation bugs.

The remainder of this section describes relevant implementation details before we evaluate SymCC in the next section. For additional documentation of low-level internals we refer interested readers to the complementary material included in the source repository at `http://www.s3.eurecom.fr/tools/symbolic_execution/symcc.html`.

4.3.1 *Compile-time instrumentation*

The instrumentation inserted by our compiler extension leaves the basic behavior of the target program unmodified; it merely enhances it with symbolic reasoning. In other words, the instrumented program still executes along the same path and produces the same effects as the original program, but additionally uses the symbolic backend to generate new program inputs that increase code coverage or possibly trigger bugs in the target program.

Since our compiler extension is implemented as an LLVM pass, it runs in the "middle-end" of LLVM-based compilers—after the frontend has translated the source language into LLVM bitcode but before the backend transforms the bitcode into machine code. SymCC thus needs to support the instructions and intrinsic functions of the LLVM bitcode language. We implement the same semantics as IR-based symbolic interpreters of LLVM bitcode, such as KLEE [12] and S2E [17]. In contrast to the interpreters, however, we do not *perform* the symbolic computations corresponding to the bitcode instructions at instrumentation time but instead generate code ahead of time that performs them during execution.[2] This means that the instrumentation step

2 This also distinguishes our approach from what the formal verification community calls *symbolic compilation* [80]. Symbolic compilers translate the entire program to a symbolic representation in order to reason about all execution paths at once, while we—like all symbolic execution systems—defer reasoning to run time, where it is necessarily restricted to a subset of all possible execution paths.

happens only once, followed by an arbitrary number of executions. Furthermore, the code that we inject is subject to compiler optimizations and eventually runs as part of the target program, without the need to switch back and forth between the target and an interpreter or attached observer. It is for this reason that we implemented the instrumentation logic from scratch instead of reusing code from KLEE or others: those systems perform run-time instrumentation whereas our implementation needs to instrument the target at compile time.

There is a trade-off in positioning SymCC's pass relative to the various optimization steps. Early in the optimizer, the bitcode is still very similar to what the frontend emitted, which is typically inefficient but relatively simple and restricted to a subset of the LLVM bitcode instruction set. In contrast, at later stages of the optimizer pipeline, dead code has been optimized away and expensive expressions (e.g., multiplication) have been replaced with cheaper ones (e.g., bit shifts); such optimized code allows for less and cheaper instrumentation but requires handling a larger portion of the instruction set. In the current implementation, our pass runs in the middle of the optimization pipeline, after basic optimizations like dead-code elimination and strength reduction but before the vectorizer (i.e., the stage that replaces loops with SIMD instructions on supported architectures). Running our code even later could improve the performance of compiled programs but would complicate our implementation by requiring us to implement symbolic handling of vector operations; we opted for implementation simplicity. It would be interesting to experiment more with the various options of positioning SymCC in the optimization pipeline; we defer such improvements to future work.

In the previous chapter, we found that symbolic execution is fastest when it executes at the level of machine code, but that SMT queries are easiest when generated based on the higher-level semantics of an intermediate representation (see Section 3.5.1). This is exactly the setup of SymCC: we reason about computations at the level of LLVM bitcode, but the injected code is compiled down to efficient machine code.

It is sometimes argued that binary-based vulnerability search is more effective than source-based techniques because it examines the instructions that the processor executes instead of a higher-level representation; it can discover bugs that are introduced during compilation. A full evaluation of this claim is outside the scope of this thesis. However, we remark that SymCC could address concerns about compiler-introduced bugs by performing its instrumentation at the very end of the optimization pipeline, just before code generation. At this point, all compiler optimizations that may introduce vulnerabilities have been performed, so SymCC would instrument an almost final version of the program—only the code-generation step needs to be trusted.

Moreover, the next chapter presents a way to apply compilation-based symbolic execution to binaries in scenarios where the need arises.

The reader may wonder whether SymCC is compatible with compiler-based sanitizers, such as address sanitizer [72] or memory sanitizer [76]. In principle, there is no problem in combining them. Recent work by Österlund et al. shows that sanitizer instrumentation can help to guide fuzzers [58]. We think that there is potential in the analogous application of the idea to symbolic execution—sanitizer checks could inform symbolic execution systems where generating new inputs is most promising. However, our current implementation, like most concolic execution systems, separates test case generation from input evaluation: sanitizers check whether the *current* input leads to unwanted behavior, while SymCC generates *new* inputs from the current one. We leave the exploration of sanitizer-guided symbolic execution in the spirit of Österlund et al. to future work.

4.3.2 *Shadow memory*

In general, we store the symbolic expressions associated with data in a shadow region in memory. Our run-time support library keeps track of memory allocations in the target program and maps them to shadow regions containing the corresponding symbolic expressions that are allocated on a per-page basis. There is, however, one special case: the expressions corresponding to function-local variables are stored in local variables themselves. This means that they receive the same treatment as regular data during code generation; in particular, the compiler's register allocator may decide to place them in machine registers for fast access.

It would be possible to replace our allocation-tracking scheme with an approach where shadow memory is at a fixed offset from the memory it corresponds to. This is the technique used by popular LLVM sanitizers [72, 76]. It would allow constant-time lookup of symbolic expressions, where currently the lookup time is logarithmic in the number of memory pages containing symbolic data. However, since this number is usually very small (in our experience, below 10), we opted for the simpler implementation of on-demand allocation.

4.3.3 *Symbolic backend*

We provide two different symbolic backends: Our own backend is a thin wrapper around Z3. It is bundled as a shared object and linked into the instrumented target program. The compiler pass inserts calls to the backend, which then constructs the required Z3 expressions and queries the SMT solver in order to generate new program inputs.

However, since the backend is mostly independent from the execution component and only communicates with it via a simple in-

terface, we can replace it without affecting the execution component, our main contribution. We demonstrate this flexibility by integrating the QSYM backend, which can optionally be used instead of our simple Z3 wrapper: We compile a shared library from the portion of QSYM that handles symbolic expressions, link it to our target program and translate calls from the instrumented program into calls to the QSYM code. The interface of our wrapper around the QSYM code consists of a set of functions for expression creation (e.g., `SymExpr _sym_build_add(SymExpr a, SymExpr b)`), as well as helper functions to communicate call context and path constraints; adding a path constraint triggers the generation of new inputs via Z3. Effectively, this means that we can combine all the sophisticated expression handling from the QSYM backend, including dependency tracking between expressions and back-off strategies for hot code paths [86], with our own fast execution component.

4.3.4 *Concreteness checks*

In Section 4.2.4, we highlighted the importance of concreteness checks: for good performance, we need to restrict symbolic reasoning (i.e., the involvement of the symbolic backend) to cases where it is necessary. In other words, when all operands of a computation are concrete, we should avoid any call to the symbolic backend. In our implementation, symbolic expressions are represented as pointers at run time, and the expressions for concrete values are null pointers. Therefore, checking the concreteness of a given expression during execution is a simple null-pointer check. Before each computation in the bitcode, we insert a conditional jump that skips symbolic handling altogether if all operands are concrete; if at least one operand is symbolic, we create the symbolic expressions for the other operands as needed and call out to the symbolic backend. Obviously, when the compiler can infer that a value is a compile-time constant and thus never symbolic at run time, we just omit the generation of code for symbolic handling.

By accelerating concrete computations during symbolic execution, we alleviate a common shortcoming of conventional implementations. Typically, only a few computations in a target program are symbolic, whereas the vast majority of operations involve only concrete values. When symbolic execution introduces a lot of overhead even for concrete computations (as is the case with current implementations despite their concreteness checks), the overall program execution is slowed down considerably. Our approach, in contrast, allows us to perform concrete computations at almost the same speed as in uninstrumented programs, significantly speeding up the analysis. Section 4.4 shows measurements to support this claim.

4.3.5 *Interacting with the environment*

Most programs interact with their environment, e.g., by working with files, or communicating with the user or other processes. Any implementation of symbolic execution needs to either define a boundary between the analyzed program and the (concrete) realm of the operating system, or execute even the operating system symbolically (which is possible in S2E [17]). QSYM [86], for example, sets the boundary at the system call interface—any data crossing this boundary is made concrete.

In principle, our approach does not dictate where to stop symbolic handling, as long as all code can be compiled with our custom compiler.[3] However, for reasons of practicality, SymCC does not assume that all code is available. Instead, instrumented code can call into any uninstrumented code at run time; the results will simply be treated as concrete values. This enables us to degrade gracefully in the presence of binary-only libraries or inline assembly, and it gives users a very intuitive way to deliberately exclude portions of the target from analysis—they just need to compile those parts with a regular compiler. Additionally, we implement a special strategy for the C standard library: we define wrappers around some important functions (e.g., `memset` and `memcpy`) that implement symbolic handling where necessary, so users of SymCC do not need to compile a symbolic version of `libc`. It would be possible to compile the standard library (or relevant portions of it) with our compiler and thus move the boundary to the system call interface, similarly to KLEE and QSYM; while this is an interesting technical challenge, it is orthogonal to the approach we present here.

4.3.6 *Supporting additional source languages*

Since SymCC uses the compiler to instrument target programs, it is in principle applicable to programs written in any compiled programming language. Our implementation builds on top of the LLVM framework, which makes it particularly easy to add support for programming languages with LLVM-based compilers, such as C++ [78], Rust [79] and Go [31]. We have implemented C++ support in SymCC, and we use it as an example for describing the generalized process of adding support for a new source language. The procedure consists of two steps, which we discuss in more detail below: (1) loading our LLVM pass into the compiler and (2) compiling the language's run-time library.

3 Our current implementation is restricted to user-space software.

Loading the pass

Any LLVM-based compiler eventually generates bitcode and passes it to the LLVM backend for optimization and code generation. In order to integrate SymCC, we need to instruct the compiler to load our compiler pass into the LLVM backend. In the case of `clang++`, the LLVM project's C++ compiler, loading additional passes is possible via the options `-Xclang -load -Xclang /path/to/pass`. Therefore, a simple wrapper script around the compiler is all that is needed. Note that the ability to load SymCC's compiler pass is the only requirement for a basic analysis; however, without instrumentation of the run-time library (detailed below), the analysis loses track of symbolic expressions whenever data passes through a function provided by the library.

Compiling the run-time library

Most programming languages provide a run-time library; it often abstracts away the interaction with the operating system, which typically requires calling C functions, and offers high-level functionality. The result of compiling it with SymCC is an instrumented version of the library that allows SymCC to trace computations through library functions. In particular, it allows the analysis to mark user input read via the source language's idiomatic mechanism as symbolic, an essential requirement for concolic execution. C++ programs, for example, typically use `std::cin` to read input; this object, defined by the C++ standard library, may rely on the C function `getc` internally, but we need an instrumented version of `std::cin` in order to trace the symbolic expressions returned by `getc` through the run-time library and into user code.

For C++ support in SymCC, we chose `libc++` [48], the LLVM project's implementation of the C++ standard library. It has the advantages that it is easy to build and that it does not conflict with `libstdc++`, the GNU implementation of the library installed on most Linux distributions. Compiling it with SymCC is a matter of setting the `CC` and `CXX` environment variables to point to SymCC before invoking the regular build scripts.

With those two steps—loading the compiler pass and compiling the run-time library—we can provide full support for a new source language.[4] As a result, SymCC ships with a script that can be used as a drop-in replacement for `clang++` in the compilation of C++ code.

4 Occasionally, frontends for new languages may emit bitcode instructions that SymCC cannot yet handle. In the case of C++, we had to add support for a few instructions that arise in the context of exception handling (`invoke`, `landingpad`, `resume`, and `insertvalue`).

4.4 EVALUATION

In this section we evaluate SymCC. We first analyze our system's performance on synthetic benchmarks (Section 4.4.1), allowing for precisely controlled experiments. Then we evaluate our prototype on real-world software (Section 4.4.2), demonstrating that the advantages we find in the benchmarks translate to benefits in finding bugs in the real world. The raw data for all figures is available at http://www.s3.eurecom.fr/tools/symbolic_execution/symcc.html.

4.4.1 *Benchmarks*

For our benchmarks we use the setup that we proposed in the previous chapter (see Section 3.4.1): at its core, it uses a set of test programs that was published in the course of the DARPA Cyber Grand Challenge (CGC), along with inputs that trigger interesting behavior in each application (called *proofs of vulnerability* or *PoVs*). The same set of programs has been used by Yun et al. in the evaluation of QSYM [86], so we know that QSYM is capable of analyzing them, which enables a fair comparison. We applied the necessary patches for KLEE in order to enable it to analyze the benchmark programs as well.[5] Note that we excluded five programs because they require inter-process communication between multiple components, making them hard to fit into our controlled execution environment, and one more, `NRFIN_00007`, because it contains a bug that makes it behave differently when compiled with different compilers (see Listing 4.5).

Listing 4.5: A bug in the code of `NRFIN_00007`. The variable `ret` is used uninitialized; if its value is non-zero, the program exits prematurely without ever reading user input. Therefore, the program's behavior effectively depends on the stack layout and previous stack contents.

```
1  int main(void) {
2      int ret;
3      size_t size;
4
5      malloc_init();
6
7      if (ret != 0)
8          _terminate(ret);
9
10     // ...
11 }
```

A major advantage of the CGC programs over other possible test sets is that they eliminate unfairness which may otherwise arise from the different instrumentation boundaries in the systems under com-

5 http://www.s3.eurecom.fr/tools/symbolic_execution/ir_study.html

parison (see Section 4.3.5): in contrast with KLEE and QSYM, SymCC does not currently execute the C standard library symbolically. It would therefore gain an unfair speed advantage in any comparison involving `libc`. The CGC programs, however, use a custom "standard library" which we compile symbolically with SymCC, thus eliminating the bias.[6]

We ran the benchmark experiments on a computer with an Intel Core i7-8550U CPU and 32 GB of RAM, using a timeout of 30 minutes per individual execution. We use SymCC with the QSYM backend, which allows us to combine our novel execution mechanism with the advanced symbolic backend by Yun et al.

Comparison with other state-of-the-art systems

We begin our evaluation by comparing SymCC with existing symbolic execution engines on the benchmark suite described above, performing three different experiments:

1. We compare *pure execution time*, i.e., running the target programs inside the symbolic execution tools but without any symbolic data.

2. We analyze *execution time with symbolic inputs*.

3. We compare the *coverage* of test cases generated during concolic execution.

The targets of our comparison are KLEE [12] and QSYM [86]. We decided for KLEE because, like SymCC, it works on LLVM bitcode generated from source code; an important difference, however, is that KLEE *interprets* the bitcode while SymCC *compiles* the bitcode together with code for symbolic processing. Comparing with KLEE therefore allows us to assess the value of compilation in the context of symbolic execution. The decision for QSYM is largely motivated by its fast execution component. Its authors demonstrated considerable benefits over other implementations, and our own work provides additional evidence for the notion that QSYM's execution component achieves high performance in comparison with several state-of-the-art systems (see Chapter 3). Moreover, our reuse of QSYM's symbolic backend in SymCC allows for a fair comparison of the two systems' execution components (i.e., their frontends). QSYM's approach to symbolic execution requires a relatively complex implementation because the system must handle the entire x86 instruction set—we demonstrate that SymCC achieves comparable or better performance with a much simpler implementation (and the additional benefit of architecture

6 The Linux port of the custom library still relies on `libc` in its implementation, but it only uses library functions that are thin wrappers around system calls without added logic, such as `read`, `write` and `mmap`. KLEE and QSYM concretize at the system-call interface, so the instrumentation boundary is effectively the same as for SymCC.

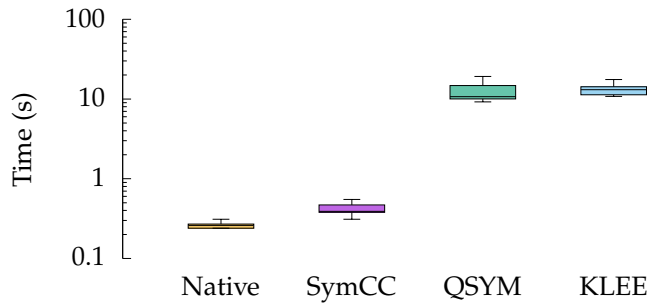

Figure 4.2: Time spent on pure execution of the benchmark programs, i.e., without symbolic data. Note the logarithmic scale of the time axis. "Native" is the regular execution time of the uninstrumented programs. On average, SymCC is faster than QSYM by 28× and faster than KLEE by 30× (KLEE can execute only 56 out of 116 programs).

independence, at the cost of requiring source code or at least LLVM bitcode).

In order to save on the already significant use of computational resources required for our evaluation, we explicitly excluded two other well-known symbolic execution systems: S2E [17] and Driller [77]. S2E, being based on KLEE, is very similar to KLEE in the aspects that matter for our evaluation, and preliminary experiments did not yield interesting insights. Driller is based on angr [74], whose symbolic execution component is implemented in Python. While this gives it distinct advantages for scripting and interactive use, it also makes execution relatively slow [86]. We therefore did not consider it an interesting target for a performance evaluation of symbolic execution.

PURE EXECUTION TIME We executed KLEE, QSYM and SymCC on the CGC programs, providing the PoVs as input. For the measurement of pure execution time, we did not mark any data as symbolic, therefore observing purely concrete execution inside the symbolic execution engines. In many real-world scenarios, only a fraction of the data in the tested program is symbolic, so efficient handling of non-symbolic (i.e., concrete) computations is a requirement for fast symbolic execution [86]. Figure 4.2 shows the results: SymCC executes most programs in under one second (and is therefore almost as fast as native execution of uninstrumented programs), while QSYM and KLEE need seconds up to minutes.

EXECUTION TIME WITH SYMBOLIC INPUTS Next, we performed concolic execution on the CGC programs, again using the PoVs as input. This time, we marked the input data as symbolic, so that symbolic execution would generate new test cases along the path dictated by each PoV. For a fair comparison, we configured KLEE to perform

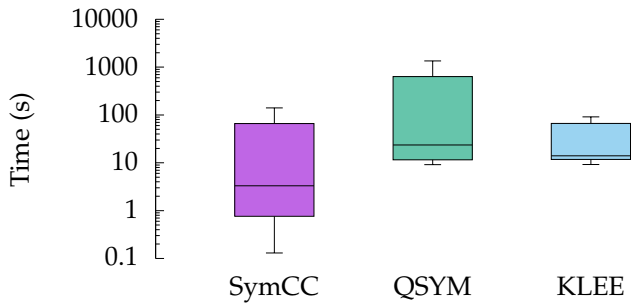

Figure 4.3: Time spent on concolic execution of the benchmark programs, i.e., with symbolic inputs (logarithmic scale). SymCC is faster than QSYM by an average factor of 10× and faster than KLEE by 12× (KLEE can execute only 56 out of 116 programs).

concolic execution like QSYM and SymCC. This setup avoids bias from KLEE's forking and scheduling components. It is worth noting, however, that KLEE still performs some additional work compared to QSYM and SymCC: since it does not rely on external sanitizers to detect bugs, it implements similar checks itself, thus putting more load on the SMT solver. Also, it features a more comprehensive symbolic memory model. Since these are intrinsic aspects of KLEE's design, we cannot easily disable them in our comparison.

In essence, all three symbolic execution systems executed the target program with the PoV input, at each conditional attempting to generate inputs that would drive execution down the alternative path. The results are shown in Figure 4.3: SymCC is considerably faster than QSYM and KLEE even in the presence of symbolic data.

COVERAGE Finally, we measured the coverage of the test cases generated in the previous experiment using the methodology of Yun et al. [86]: for each symbolic execution system, we recorded the combined coverage of all test cases per target program in an AFL coverage map [88].[7] On each given target program, the result was a set of covered program points for each system, which we will call S for SymCC and R for the system we compare to (i.e., KLEE or QSYM). We then assigned a score d in the range $[-1.0, 1.0]$ as per Yun et al. [86]:

$$d(S,R) = \begin{cases} \frac{|S-R|-|R-S|}{|(S \cup R)-(S \cap R)|} & \text{if } S \neq R \\ 0 & \text{otherwise} \end{cases}$$

Intuitively, a score of 1 would mean that SymCC covered all program paths that the other system covered and some in addition, whereas a score of -1 would indicate that the other system reached all the

7 Traditional coverage measurement, e.g., with `gcov`, does not work reliably on the CGC programs because of the bugs that have been inserted.

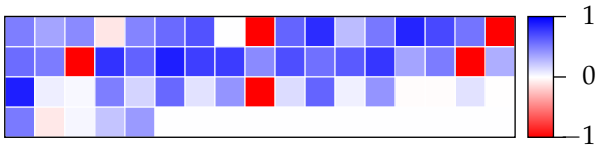

Figure 4.4: Coverage score comparing SymCC and KLEE per tested program (visualization inspired by Yun et al. [86]): blue colors mean that SymCC found more paths, red colors indicate that KLEE found more, and white symbolizes equal coverage. SymCC performs better on 46 programs and worse on 10 (comparison restricted to the programs that KLEE can execute, i.e., 56 out of 116).

paths covered by SymCC plus some more. We remark that this score, while giving a good intuition of relative code coverage, suffers from one unfortunate drawback: It does not put the coverage difference in relation with the overall coverage. In other words, if two systems discover exactly the same paths except for a single one, which is only discovered by one of the systems, then the score is extreme (i.e., 1 or -1), no matter how many paths have been found by both systems. In our evaluation, the coverage difference between SymCC and the systems we compare to is typically small in comparison to the overall coverage, but the score cannot accurately reflect this aspect. However, for reasons of comparability we adopt the definition proposed by Yun et al. unchanged; it still serves the purpose of demonstrating that SymCC achieves similar coverage to other systems in less time.

We visualize the coverage score per test program in Figures 4.4 and 4.5. The former shows that SymCC generally achieves a higher coverage level than KLEE; we mainly attribute differences to the significantly different symbolic backends. The latter demonstrates that SymCC's coverage is comparable to QSYM's, i.e., the compilation-based execution component provides information of comparable quality to the symbolic backend. We suspect the reason that coverage of some programs differs at all—despite the identical symbolic backends in QSYM and SymCC—is twofold:

1. SymCC derives its symbolic expressions from higher-level code than QSYM (i.e., LLVM bitcode instead of x86 machine code). This sometimes results in queries that are easier for the SMT solver, leading to higher coverage.

2. On the other hand, the lower-level code that QSYM analyzes can lead to test cases that increase coverage of the program under test at the machine-code level.

We conclude that compilation-based symbolic execution is significantly faster than IR-based and even IR-less symbolic execution in our benchmarks while achieving similar code coverage and maintaining a simple implementation.

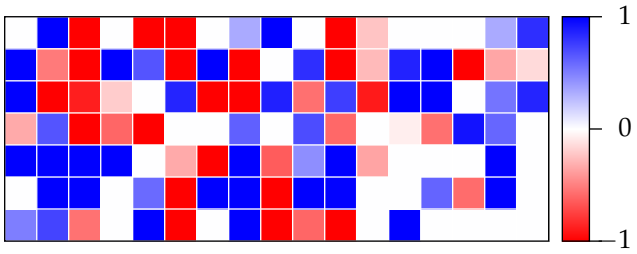

Figure 4.5: Comparison of coverage scores between SymCC and QSYM. SymCC found more paths on 47 programs and less on 40; they discovered the same paths on 29 programs. Similar coverage is expected because SymCC uses the same symbolic backend as QSYM.

Initialization overhead

In the course of our evaluation we noticed that QSYM and KLEE have a relatively large constant-time overhead in each analysis. For example, on our test machine, QSYM always runs for several seconds, independently of the program under test or the concreteness of the input. The overhead is presumably caused by costly instrumentation work performed by the symbolic executor at the start of the analysis (something that SymCC avoids by moving instrumentation to the compilation phase). Therefore, we may assume that the execution times T_{SymCC} and T_{other} are not related by a simple constant speedup factor but can more accurately be represented via initialization times I_{SymCC} and I_{other}, analysis times A_{SymCC} and A_{other}, and a speedup factor S that only applies to the analysis time:

$$T_{\text{SymCC}} = I_{\text{SymCC}} + A_{\text{SymCC}} \tag{4.1}$$

$$T_{\text{other}} = I_{\text{other}} + A_{\text{other}} = I_{\text{other}} + S \cdot A_{\text{SymCC}} \tag{4.2}$$

Consequently, we can compute the speedup factor as follows:

$$S = \frac{T_{\text{other}} - I_{\text{other}}}{T_{\text{SymCC}} - I_{\text{SymCC}}} \tag{4.3}$$

In order to obtain accurate predictions for the analysis time of *long-running* programs, we therefore need to take the initialization time into account when computing the speed-up factor. As a simple approximation for the worst case from SymCC's point of view, we assumed that the shortest observed execution consists of initialization only, i.e., suppose A_{SymCC} and A_{other} are zero in the analysis of the fastest-running program. In other words, for each system we subtracted the time of the fastest analysis observed in Section 4.4.1 from all measurements. Then we recomputed the speedup in the affine model presented above. For concolic execution with KLEE, we obtained an average factor of 2.4 at a constant-time overhead of 9.20 s, while for QSYM we computed a

factor of 2.7 at a constant-time overhead of 9.15 s. SymCC's constant-time overhead is 0.13 s; this confirms the benefit of instrumenting the target at compile time.

Note that this model is only relevant for long-running programs, which are rarely fuzzed.[8] Otherwise, execution time is dominated by the startup overhead of QSYM and KLEE. Nevertheless, the model shows that SymCC's performance advantage is not simply due to a faster initialization—even when we account for constant-time overhead at initialization and overestimate it in favor of QSYM and KLEE, SymCC is considerably faster than both.

Compilation time and binary size

SymCC modifies the target program extensively during compilation, which results in increased compilation time and larger binaries (because of inserted instrumentation). In order to quantify this overhead, we first compiled all 116 CGC programs with both SymCC and regular `clang`, and measured the total build time in either case. Compilation required 602 s with SymCC compared to 380 s with `clang`; this corresponds to an increase of 58 %. Note that this is a one-time overhead: once a target program is built, it can be run an arbitrary number of times.

Next, we compared the size of each instrumented executable produced by SymCC with the corresponding unmodified executable emitted by `clang`. On average, our instrumented binaries are larger by a factor of 3.4. While we have not optimized SymCC for binary size, we believe that there is potential to reduce this factor if needed. The largest contribution to code size comes from run-time concreteness checks; if binary size became a major concern, one could disable concreteness checks to trade execution time for space. In our tests we have not experienced the necessity.

Impact of concreteness checks

In Section 4.2.4, we claimed that considerable improvements can be gained by checking data for concreteness at run time and skipping symbolic computation if all operands are concrete.

To illustrate this claim, let us examine just the initialization phase of the CGC program `CROMU_00001`. During the startup, the CGC "standard library" populates a region in memory with pseudo-random data obtained by repeated AES computations on a seed value; this happens before any user input is read. In the uninstrumented version of the program, the initialization code executes within roughly 8 ms. This is the baseline that we should aim for. However, when we run a version of SymCC *with concreteness checks disabled* on `CROMU_00001`, execution

8 The documentation of AFL, for example, recommends that target programs should be fast enough to achieve "ideally over 500 execs/sec most of the time" [88].

takes more than five minutes using our own simple backend, and with the faster QSYM backend SymCC still requires 27 s. The reason is that the instrumented program calls into the symbolic backend at every operation, which creates symbolic expressions, regardless of the fact that all operands are fully concrete. The QSYM backend performs better than our simple backend because it can fold constants in symbolic expressions and has a back-off mechanism that shields the solver against overload [86]. However, recall that we are executing on concrete data only—it should not be necessary to invoke the backend at all!

In fact, concreteness checks can drastically speed up the analysis by entirely freeing the symbolic backend from the need to keep track of concrete computations. With concreteness checks enabled (as described in Section 4.3.4), the symbolic backend is only invoked when necessary, i.e., when at least one input to a computation is symbolic. For the initialization of `CROMU_00001`, enabling concreteness checks results in a reduction of the execution time to 0.14 s with the QSYM backend (down from 27 s). The remaining difference with the uninstrumented version is largely due to the overhead of backend initialization and memory operations for book-keeping.

We assessed the effect across the CGC data set with PoV inputs and found that the results confirm our intuition: concreteness checks are beneficial in almost all situations. The only 3 cases where they increased the execution time instead of decreasing it were very long-running programs that perform heavy work on symbolic data.

4.4.2 *Real-world software*

We have shown that SymCC outperforms state-of-the-art systems in artificial benchmark scenarios. Now we demonstrate that these findings apply as well to the analysis of real-world software. In particular, we show that SymCC achieves comparable or better overall performance despite its simple implementation and architecture-independent approach.

We used QSYM and SymCC in combination with the fuzzer AFL [88] to test popular open-source projects (using AFL version 2.56b); KLEE is not applicable because of unsupported instructions in the target programs. For each target program, we ran an AFL master instance, a secondary AFL instance, and one instance of either QSYM or SymCC. The symbolic execution engines performed concolic execution on the test cases generated by the AFL instances, and the resulting new test cases were fed back to the fuzzers. Note that this is a relatively naive integration between symbolic execution and fuzzer; however, since the focus of this work is on the performance of symbolic execution, we leave the exploration of more sophisticated coordination techniques to future work (see Chapter 6).

	OpenJPEG		libarchive		tcpdump	
	SymCC	QSYM	SymCC	QSYM	SymCC	QSYM
Execution time (s)	1.9	14.9	1.6	19.1	0.3	27.1
Solver time (s)	26.4	15.7	0.2	1.8	0.3	8.2
Total time (s)	28.3	30.6	1.8	20.9	0.6	35.3
Execution (%)	6.7	48.7	91.7	91.2	41.7	76.8
SMT solving (%)	93.3	51.3	8.3	8.8	58.3	23.2
Factor vs QSYM	1.1		11.6		58.8	

Table 4.1: Average time split between execution and SMT solving. See Figure 4.7 for a visualization of the total analysis times. Note how the speedup factor in the last row correlates with SymCC's improved coverage displayed in Figure 4.6.

Fuzzing is an inherently randomized process that introduces a lot of variables outside our control. Following the recommendations by Klees et al. [44], we therefore let the analysis continue for 24 hours, we repeated each experiment 30 times, and we evaluated the statistical significance of the results using the Mann-Whitney U test. Our targets are OpenJPEG, which we tested in an old version with known vulnerabilities, and the latest master versions of libarchive and tcpdump. In total, we spent 3 experiments $\times$ 2 analysis systems $\times$ 30 $\frac{\text{iterations}}{\text{experiment}\cdot\text{analysis system}} \times 3\frac{\text{CPU cores}}{\text{iteration}} \times$ 24 hours = 12 960 CPU core hours $\approx$ 17.8 CPU core months. The hardware used for these experiments was an Intel Xeon Platinum 8260 CPU with 2 GB of RAM available to each process (AFL, QSYM or SymCC).

While running the fuzzer and symbolic execution as specified above, we measured the code coverage as seen by AFL[9] (Figure 4.6) and the time spent on each symbolic execution of the target program (Table 4.1 and Figure 4.7). We found that SymCC not only executes faster than QSYM (which is consistent with the benchmarks of Section 4.4.1) but also reaches significantly higher coverage on all three test programs. Interestingly, the gain in coverage appears to be correlated with the speed improvement, which confirms our intuition that accelerating symbolic execution leads to better program testing.

Since we used an old version of OpenJPEG known to contain vulnerabilities, we were able to perform one more measurement in this case: the number of crashes found by AFL. Unfortunately, crash triage is known to be challenging, and we are not aware of a generally accepted approach to determine uniqueness. We therefore just remark that there is no significant difference between the number of AFL

9 AFL's coverage map is known to be prone to collisions and therefore does not reflect actual code coverage [29]. However, AFL bases its decisions on the coverage map, so the map is what counts when evaluating the benefit of a symbolic execution system for the fuzzer.

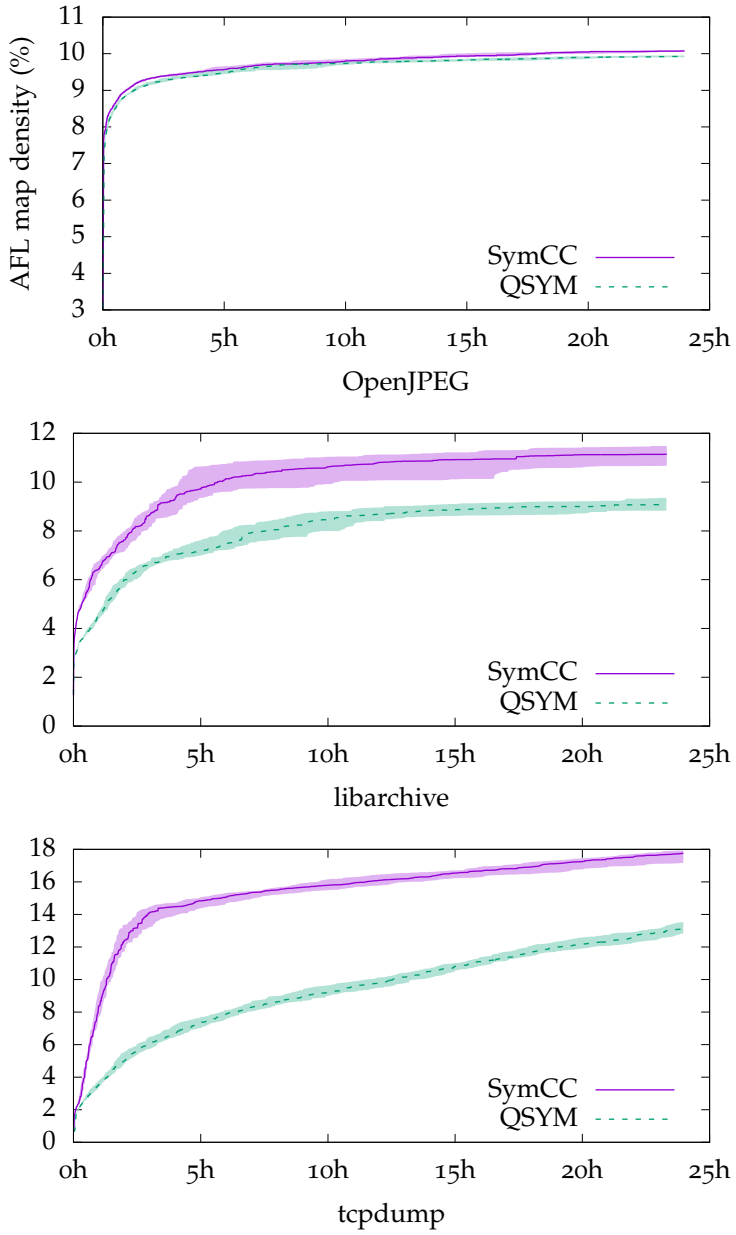

Figure 4.6: Density of the AFL coverage map over time. The shaded areas are the 95 % confidence corridors. The respective differences between QSYM and SymCC are statistically significant with $p < 0.0002$. Note that the coverage improvement correlates with the speedup displayed in Figure 4.7.

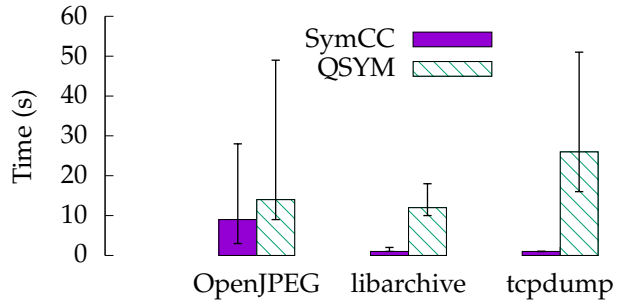

Figure 4.7: Time per symbolic execution (median and quartiles, excluding executions that exhausted time or memory resources). The difference between QSYM and SymCC is statistically significant with $p < 0.0001$. Note the correlation between higher speed here and increased coverage in Figure 4.6.

"unique crashes" found with QSYM and SymCC on this version of OpenJPEG.

In the course of our experiments with OpenJPEG, SymCC found two vulnerabilities that affected the latest master version at the time of writing as well as previously released versions. Both vulnerabilities were writing heap buffer overflows and therefore likely exploitable. They had not been detected before, even though OpenJPEG is routinely fuzzed with state-of-the-art fuzzers and considerable computing resources by Google's OSS-Fuzz project. We reported the vulnerabilities to the project maintainers, who confirmed and fixed both. The vulnerabilities were subsequently assigned CVE identifiers 2020-6851 and 2020-8112 and given high impact scores by NIST (7.5 and 8.8, respectively). In both cases, the problems arose from missing or incorrect bounds checks—symbolic execution was able to identify the potential issue and solve the corresponding constraints in order to generate crashing inputs. In the same experiments, QSYM did not find new vulnerabilities.

In conclusion, our experiments show that SymCC is not only faster than state-of-the-art systems on benchmark tests—we demonstrated that the increased speed of symbolic execution also translates to better performance when testing real-world software.

4.5 DISCUSSION AND FUTURE WORK

In this section, we discuss the results of our evaluation and show some directions for future work.

4.5.1 *Benefits of compilation*

We have seen in that our compilation-based approach provides a much faster execution component for symbolic execution than existing

IR interpreters and IR-less systems. At the same time, we retain the flexibility that comes with building symbolic execution on top of an intermediate representation (i.e., our implementation is not tied to a particular machine architecture) and the robustness of IR-less systems (i.e., computations that we cannot analyze are still performed correctly by the CPU). We believe that compilation-based symbolic execution, where applicable, has the potential of accelerating symbolic execution to a level that is comparable with fuzzing, making it significantly more useful for bug discovery and rendering the combination of symbolic execution and fuzzing even more attractive.

4.5.2 *Portability and language support*

Our current prototype supports programs written in C and C++. However, since we build on the LLVM framework, we could support any program that is translatable to LLVM bitcode. In particular, this means that we can integrate SymCC into any LLVM-based compiler, such as the default compilers for Rust [79] and Swift [1], and the alternative Go compiler `gollvm` [31]. Similarly, we can generate binaries for any machine architecture that LLVM supports, without any changes in our code. More generally, the technique of compilation-based symbolic execution applies to any compiled programming language.

4.5.3 *Binary analysis*

So far, we have only discussed compilation-based symbolic execution in contexts where the source code of the program under test is available. A common criticism of source-based tools is that they fall short when the source for parts or all of a program is not available. For example, developers may be in control of their own source code but rely on a third-party library that is available in binary form only. SymCC handles such situations by treating binary-only components as black boxes returning concrete values. While this should be sufficient for simple cases like binary-only libraries or inline assembly, there are situations where *symbolic* execution of binary-only components is necessary, i.e., where one wants to keep track of the computations inside the black boxes. We see two promising avenues for addressing such use cases with SymCC:

Lifting

SymCC currently uses compiler frontends to create LLVM bitcode from source code, but there is no fundamental reason for creating the bitcode from the source: S2E [17] popularized the idea of generating a high-level IR from binaries for the purpose of symbolic execution. It generates LLVM bitcode from the internal program representation

of QEMU [5] and runs it in KLEE [12]. A similar approach is used by angr [74], which dynamically generates VEX IR for a symbolic interpreter from binaries. Several other such *lifters* have been designed for purposes outside the realm of symbolic analysis [42]. While the IR obtained from binaries is more verbose (see Section 3.4.3), SymCC could be used in combination with a lifter to compile symbolic handling into existing binaries. Trail of Bits has recently applied a similar lifting technique to KLEE, essentially converting it from a source-based tool to a symbolic execution engine that can work on binaries [82].

In Chapter 5, we present SymQEMU, a binary-only symbolic executor that builds on the idea of combining lifting and compilation-based symbolic execution.

Hybrid with QSYM

It may be possible to combine our compilation-based approach with QSYM's capabilities of working on binaries; basically, one would benefit from SymCC's fast execution in the parts of the program under test for which source code is available and fall back to QSYM's slower observer-based approach in binary-only parts. Considering that SymCC can already work with QSYM's symbolic backend, symbolic expressions could be passed back and forth between the two realms—the main challenge then lies in handling the transitions between source-based and binary-only program components.

We would like to remark, however, that even binary-based symbolic execution is often evaluated on open-source software, and many gray-box fuzzers like AFL [88] only reach their full performance when the source code of the program under test is available for instrumentation.

4.6 RELATED WORK

As a program analysis technique, symbolic execution exists on a spectrum. On the one extreme of that spectrum, bounded model checking inlines all functions, unrolls loops up to a certain bound and translates the entire program into a set of constraints [26, 65]. While this process is sometimes called "symbolic compilation" [6], it is not to be confused with our compilation-based symbolic execution: bounded verification reasons about all executions at once, thus allowing for very sophisticated queries but pushing most of the load to the SMT solver. Our approach, in contrast, follows the tradition of symbolic execution by reasoning about the program per execution path [12, 17, 74]. On the other end of the spectrum, fuzz testing executes target programs with very light or no instrumentation, heuristically mutating inputs (and possibly using feedback from the instrumentation) in the hope of finding inputs that evoke a certain behavior, typically program crashes [7, 16, 25, 47, 88].

While bounded verification provides powerful reasoning capabilities, fuzzing is extremely fast in comparison. Conventional symbolic execution lies between the two [12, 17, 74], with state-merging approaches [46, 80] leaning more towards bounded verification, and hybrids with fuzzing attempting to produce fast but powerful practical systems [77, 86]. It is this last category of systems that forms the basis for our approach: we target a combination of symbolic execution and fuzzing similar to Driller [77] and QSYM [86]. By speeding up symbolic execution, we aim to make its more sophisticated reasoning available in situations where previously only fuzzing was fast enough.

Current work in symbolic execution, as outlined above and referenced throughout the thesis, applies either interpreter- or observer-based techniques. While early systems embedded symbolic reasoning directly [13, 32, 71], they performed the instrumentation at the level of C code, which severely restricts the set of possible input programs and complicates the implementation significantly [32]. The approach of instrumenting the program under test directly was abandoned in KLEE [12], and subsequent work in symbolic execution mostly followed its lead. We are not aware of any performance comparison between the direct embedding implemented in early work and the interpreter approach to symbolic execution implemented by KLEE and later systems; we assume that the switch happened because interpreters are more flexible and easier to implement correctly. With SymCC, we demonstrate that directly embedding concolic execution into the target program yields much higher performance than state-of-the-art systems; at the same time, however, performing the embedding at the level of the compiler's intermediate representation allows us to maintain the flexibility that is common in modern implementations.

The most closely related project outside the field of symbolic execution is Rosette, a "solver-aided programming language" [80]. It allows programmers to express symbolic constraints while writing a program, which it then executes in a "Symbolic Virtual Machine". In contrast to our approach, it is not meant for the analysis of arbitrary programs but rather aims to support the development of program-synthesis and verification tools. It requires the developer to use a domain-specific language and design the program for symbolic analysis from the start. Moreover, it does not compile the program to machine code but rather executes it in a host environment, similarly to how KLEE orchestrates multiple execution states in a single process.

SMT Kit [36] is a project that performs a similar embedding into C++, and there is (incomplete) support for automatically transforming source code to use the library [35]. The idea, if fully executed, may have led to a system similar to SymCC, but the project seems to have been abandoned years ago without a publication, and we have been unable to contact the author. We anticipate that a robust source-to-source translation would have been much more difficult to implement

than our IR transformation due to the complexity of the C++ language in comparison with LLVM bitcode. Moreover, the system would have been inherently limited to a single programming language, just like the early implementations for C mentioned above, while SymCC's transformation at the IR level allows it to support any source language for which an LLVM-based compiler exists.

4.7 CONCLUSION

We have presented SymCC, a symbolic execution system that embeds symbolic processing capabilities in programs under test via a compiler. The evaluation shows that the direct embedding yields significant improvements in the execution speed of the target programs, outperforming current approaches by a large margin. Faster execution accelerates the analysis at large and increases the chances of bug discovery, leading us to find two high-impact vulnerabilities in a heavily tested library. By using a compiler to insert symbolic handling into target programs, we combine the advantages of IR-based and IR-less symbolic execution: SymCC is architecture-independent and can support various programming languages with little implementation effort (like IR-based approaches), but the analysis is very fast—considerably faster even than current IR-less techniques.

AVAILABILITY

SymCC is publicly available at `http://www.s3.eurecom.fr/tools/symbolic_execution/symcc.html`. The page also contains links to the source code of all programs that we used in our evaluation, as well as the raw results of the experiments. SymCC's code base is thoroughly documented in order to serve the community as a basis for future research.

5

COMPILING SYMBOLIC EXECUTION INTO BINARIES

We have shown in the previous chapter that compilation-based symbolic execution can improve the performance of symbolic execution significantly when source code is available. In this chapter, we demonstrate a novel technique to enable compilation-based symbolic execution of binaries (i.e., without the need for source code). Our system, SymQEMU, builds on top of QEMU, hooking into its QEMU's dynamic binary translation to modify the intermediate representation of the target program before translating it to the host architecture. This enables SymQEMU to compile symbolic-execution capabilities into binaries and reap the associated performance benefits while maintaining architecture independence.

We present our approach and implementation, and we show that it outperforms the state-of-the-art binary symbolic executors S2E and QSYM with statistical significance; on some benchmarks, it even achieves better performance than the source-based SymCC. Moreover, our tool has found a previously unknown vulnerability in the well-tested libarchive library, demonstrating its utility in testing real-world software.

5.1 INTRODUCTION

An important characteristic of symbolic execution systems is whether they require the *source code* of the program under test (like SymCC, presented in the previous chapter) or instead apply to *binary-only* programs in a black-box fashion. While source-based testing is sufficient when one is testing one's own products or open-source software, many real-world scenarios require the ability to analyze binaries without the source code available:

- We are increasingly surrounded by and rely upon embedded devices. Their firmware is typically available in binary form only. Security audits therefore require binary-analysis tools [73, 87].
- Even when testing one's own products, proprietary library dependencies may not ship with source code, rendering source-based approaches infeasible.
- Source-based testing may simply be impractical for large programs under test. With a source-based tool, one typically needs to build all library dependencies in a dedicated manner prescribed by the tool, which may put a large burden on the tester.

> Moreover, if the program under test is implemented in a mix of programming languages, chances are that source-based tools cannot handle all of them.

When a binary-only symbolic executor is called for, users often face a dilemma: tools optimize either for performance or for architecture independence but rarely provide both. For example, QSYM [86] has shown how to implement very fast symbolic execution of binaries, but it achieves its high speed by tying the implementation to the instruction set of x86 processors. Not only does this render the system architecture-dependent, it also increases its complexity due to the sheer size of modern processors' instruction sets; in the authors' own words, their approach is to "pay for the implementation complexity to reduce execution overhead". In contrast, S2E [17] is an example of a system that is broadly applicable yet suffers from relatively low execution speed. S2E can conceptually analyze code for most CPU architectures, including kernel code. However, its wide applicability is bought with multiple translations and finally interpretation of the target program (to be detailed later), which increase the system's complexity and ultimately affect performance. In fact, it appears that high performance in binary-only symbolic analysis is often achieved with highly specialized implementations—a design choice that is in conflict with architectural flexibility.

In this chapter, we show an alternative that (a) is independent of the target architecture of the program under test, (b) has low implementation complexity, yet (c) achieves high performance. The key insight of our system, SymQEMU, is that the CPU emulation of QEMU [5], based on dynamic binary translation, can be combined with compilation-based symbolic execution (see Chapter 4): instead of performing a computationally expensive translation of the target program to an intermediate representation that is subsequently interpreted symbolically (like in S2E), we hook into QEMU's binary-translation mechanism in order to compile symbolic handling directly into the machine code that the emulator emits and executes. This approach yields performance superior to state-of-the-art systems while retaining full platform independence. Currently, we focus on Linux user-mode programs (i.e., ELF binaries), but it would be possible to extend the concept to full-system emulation for arbitrary QEMU-supported platforms (e.g., for firmware analysis). Moreover, we will make SymQEMU publicly available to foster future research in the area.

We compared SymQEMU to state-of-the-art binary symbolic executors S2E and QSYM, and found that it outperforms both in terms of coverage reached over time. Moreover, we show that SymQEMU's performance is similar to that of SymCC, even though the latter requires access to source code (see Chapter 4). Finally, we submitted SymQEMU to Google FuzzBench, a comparison framework for fuzzers;

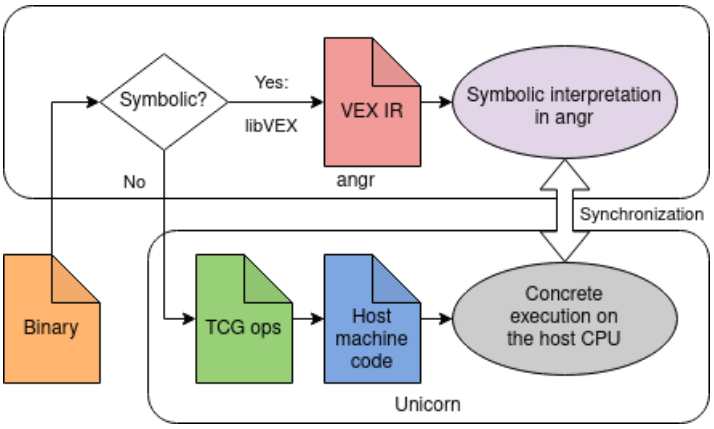

Figure 5.1: Overview of angr: the target program is lifted to VEX IR and interpreted symbolically or executed concretely inside the Unicorn CPU emulator.

although the test suite is not targeted at symbolic execution systems, SymQEMU outperformed all included fuzzers on 3 out of 21 targets.

In summary, this chapter makes the following contributions:

- We analyze state-of-the-art implementations of binary-only symbolic execution and identify the respective strengths and weaknesses of their designs.
- We present an approach that combines the strengths of existing systems while avoiding most of their weaknesses; the core idea is a novel technique to apply compilation-based symbolic execution to binaries.
- We evaluate our system in Google FuzzBench, as well as on open-source and closed-source real-world software.

5.2 THE STATE OF THE ART

We now describe three popular state-of-the-art implementations and study the design choices with which they address the challenges of binary-only symbolic execution (see Section 2.3).

5.2.1 *Angr [74]*

A "classic" translating symbolic executor. It reuses VEX, the intermediate language and translator of the Valgrind framework [56]. The target programs are translated at run time; the symbolic executor then interprets the VEX instructions. As an optimization, angr can execute computations that do not involve symbolic data (i.e., whose results do not depend on program input) in Unicorn [66], a fast CPU emulator based on QEMU [5]. Figure 5.1 illustrates the design.

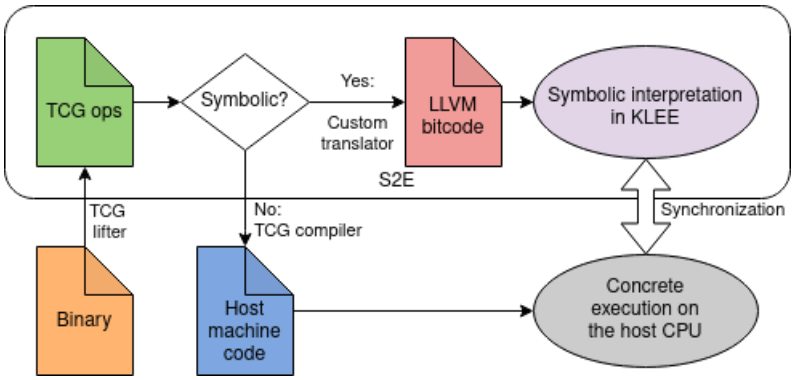

Figure 5.2: Overview of S2E: the target program is lifted to TCG ops and then either translated to host machine code or lifted once more and executed symbolically in KLEE.

By virtue of being based on VEX, angr inherits support for all architectures that VEX knows how to handle. Since the core of the symbolic executor is written in Python, it is rather slow (see Chapter 3) but very versatile.

5.2.2 *S2E [17]*

Created from the desire to extend the reach of the source-based symbolic-execution system KLEE [12] to the target program's dependencies and the operating-system kernel. To this end, S2E runs an entire operating system inside the emulator QEMU [5] and connects it to KLEE in order to execute relevant code symbolically (see Figure 5.2). The resulting system is rather complex, involving multiple translations of the program under test:

1. QEMU is a binary translator, i.e., in normal operation, it translates the target program from machine code to an intermediate representation (called *TCG ops*), then recompiles it to machine code for the host CPU.

2. When computations involve symbolic data, the modified QEMU used by S2E does not recompile the TCG ops to host code; instead, it translates them to LLVM bitcode [49], which is subsequently passed to KLEE.

3. KLEE interprets the LLVM bitcode symbolically and hands the concrete portion of the results back to QEMU.

This approach results in a very flexible system that can conceptually handle many different architectures and trace computations through all layers of the operating system.[1] However, the flexibility comes at a

1 At the time of writing, only x86 is fully supported (https://github.com/S2E/s2e-env/issues/268).

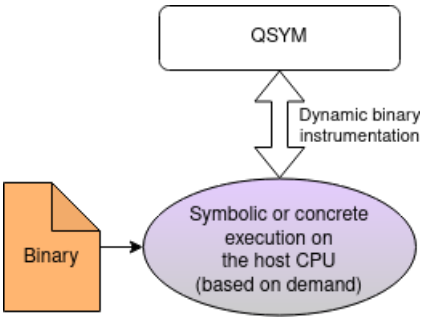

Figure 5.3: Overview of QSYM: the target program is executed directly on the CPU while QSYM instruments it dynamically.

cost: S2E is a complicated system with a large code base. Moreover, the two-step translation from machine code to TCG ops and from there to LLVM bitcode hurts its performance (see Chapter 3). Compared with angr from a user's point of view, S2E is more involved to set up and run but provides a more comprehensive analysis.

5.2.3 *QSYM [86]*

With a strong emphasis on performance, QSYM does not translate the target program to an intermediate language. Instead, it instruments x86 machine code at run time to add symbolic tracing to binaries (see Figure 5.3). Concretely, it employs Intel Pin [52], a dynamic binary instrumentation framework, to insert hooks into the target program. Inside the hooks, it performs the symbolic equivalent of the machine-code instructions that the program executes.

This design yields a very fast and robust symbolic executor for x86 programs. However, the system is inherently restricted to a single target architecture, and the implementation is tedious because it needs to handle each and every x86 instruction that can be expected to occur in relevant computations. In previous work, we have found QSYM to be a great tool for the analysis of x86 binaries, but adding support for another architecture would be a significant amount of work.

5.3 RELATION TO SYMCC

We consider it worthwhile to point out how our work on binary-only symbolic execution relates to SymCC, presented in the previous chapter, and why there is a need for yet another symbolic executor.

In short, SymCC does not work on binaries. Its compilation-based approach fundamentally requires a compiler—SymCC is therefore applicable only when source code of the program under test is available. Nonetheless, we considered the approach promising enough to search for a way to apply it to binary-only symbolic execution. A major contribution of this chapter is to demonstrate how compilation-

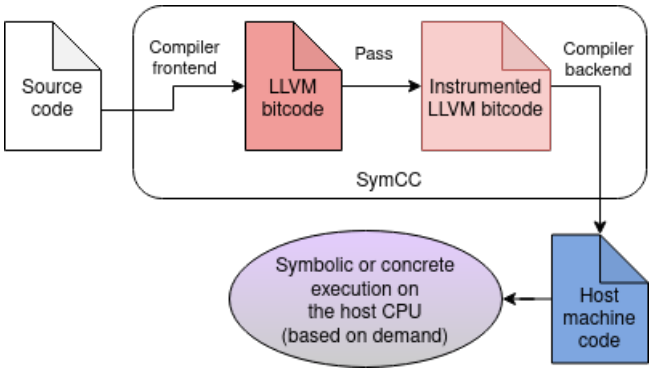

Figure 5.4: Overview of SymCC: the source code of the target program is compiled to machine code; symbolic handling is injected at the level of LLVM bitcode in the compiler.

based symbolic execution can, in fact, be made to work efficiently on binaries.

Figure 5.4 recalls the design of SymCC for comparison with the figures in Section 5.2.

5.4 SYMQEMU

We now present the design and implementation of our binary-only symbolic executor SymQEMU. It draws from previous work and combines the advantages of state-of-the-art systems with novel ideas to create a fast yet flexible analysis engine.

5.4.1 *Design*

The system has two main goals:

1. Achieve high performance in order to scale to real-world software.
2. Stay reasonably platform-independent, i.e., adding support for a processor architecture should not require a major effort.

Based on the survey in Section 5.2, we observe that popular state-of-the-art systems typically achieve one of those goals, but not both: among those presented, S2E and angr are highly flexible yet fall behind in performance (as shown in Chapter 3), whereas QSYM is very fast but intimately tied to the x86 platform [86].

We have seen that current solutions which are platform-independent translate the program under test to an intermediate representation—this way, in order to support a new architecture, only the translator has to be ported. Ideally, one picks an intermediate language for which translators from many relevant architectures exist already. Representing programs in an architecture-independent way for flexibility is a

well-known technique that has been successfully applied in many other domains, such as compiler design [49] and static binary analysis [42]. We therefore incorporate it into our design as well.

While translating programs to an intermediate representation gives us flexibility, we need to be aware of the impact on performance: translating binary-only programs statically is challenging because disassembly may not be reliable (especially in the presence of indirect jumps [60]), and performing the translation at run time incurs overhead during the analysis. We believe that this is the core reason why translating symbolic executors like S2E and angr lag behind non-translating systems like QSYM in terms of performance. Our goal is to find a way to build a translating system that still performs well.

First, we note that the speed of both S2E and angr is affected by non-essential issues that could be fixed with an engineering effort:

- S2E translates the program under test twice (see Section 5.2.2). The second translation could be avoided if symbolic execution was implemented on the first intermediate representation.[2]

- Angr's performance suffers from the Python implementation; porting the core to a faster programming language would likely result in a noteworthy speedup.

However, our contribution goes beyond just identifying and avoiding those two problems. This is where a second observation comes into play: Both S2E and angr, as well as all other translating binary-only symbolic executors that we are aware of, *interpret* the intermediate representation of the program under test. (This is independent of the modifications suggested above—interpretation is a core part of their design.) We conjecture that *compiling* an instrumented version of the target program yields much higher performance. SymCC shows that this is true of source-based symbolic execution (see Chapter 4), but its compiler-based design inherently requires source code and therefore doesn't apply to the binary-only use case.

Our approach, inspired by the above observations, is the following:

1. Translate the target program to an intermediate language at run time.

2. Instrument the intermediate representation as necessary for symbolic execution.

3. Compile the intermediate representation to machine code suitable for the CPU running the analysis and execute it directly.

By compiling the instrumented target program to machine code, we compensate for the performance penalty incurred by translating the

2 In fact, the developers of S2E have plans to do just that, documented at https://github.com/S2E/s2e-env/issues/178.

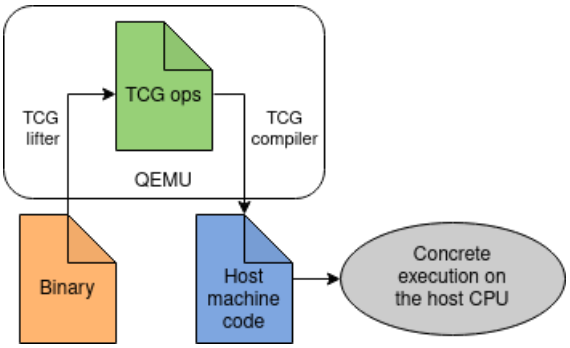

Figure 5.5: Overview of regular QEMU: the target program is translated to TCG ops, which are subsequently compiled to machine code and executed on the host CPU.

binary to an intermediate language in the first place: The CPU executes machine code much faster than an interpreter can run the intermediate representation, such that we achieve performance comparable to a non-translating system while retaining the advantage of architecture independence that comes with program translation.

5.4.2 *Implementation*

We implemented SymQEMU on top of QEMU [5], as suggested by the name. We have chosen QEMU because it is a robust system emulator that supports a plethora of architectures. Building on it, we are able to achieve our goal of platform independence. Note that S2E is similarly based on QEMU, presumably for similar reasons. But there is another characteristic of QEMU that caters to our needs and differentiates it from other translators: QEMU does not only translate binaries to a processor-independent intermediate representation, it also has facilities for compiling the intermediate language down to machine code for the host CPU. We leverage this mechanism to achieve our second goal: performance.

Note that the Valgrind framework supports a similar mechanism, which its authors call "disassemble-and-resynthesize" [56]; the main advantage of QEMU over Valgrind for our purposes is that QEMU can translate binaries from a given guest architecture into machine code for a *different* host architecture, as well as emulate an entire system, which makes it a better basis for future extensions supporting cross-architecture firmware analysis.

Concretely, we extend a component in QEMU called *Tiny Code Generator (TCG)*. In unmodified QEMU, TCG is responsible for translating blocks of guest-architecture machine code to an architecture-independent language called TCG ops, then compile those TCG ops to machine code for the host architecture (see Figure 5.5). The translated blocks are subsequently cached for performance reasons, so transla-

tion needs to happen only once per execution. SymQEMU inserts one more step into the process: While the program under test is being translated to TCG ops, we emit not only the instructions that emulate the guest CPU but also additional TCG ops to construct symbolic expressions for the results (see Figure 5.6).

For example, suppose that a function in a target program adds the constant 42 to an input integer (using C code for the example):

```
int add42(int x) {
  return x + 42;
}
```

With optimization enabled, GCC inlines the function and translates it to this assembly instruction when compiling for the x86-64 architecture:

```
lea    esi,[rax+0x2a]
```

The machine code is all that SymQEMU gets; it does not have access to the source code (which we display for illustration purposes only). When we execute the target, TCG produces the following architecture-independent representation of the machine code:

```
movi_i64 tmp12, $0x2a
add_i64 tmp2, rax, tmp12
ext32u_i64 rsi, tmp2
```

Note that the arguments of TCG ops are ordered like x86 assembly in Intel syntax, i.e., the destination is the first argument of any instruction. The instructions above perform a 64-bit addition and store the result as a 32-bit integer. Regular QEMU would translate these TCG ops to machine code for the host architecture. SymQEMU, however, inserts additional instructions for symbolic computation before the code is translated to the host architecture:

```
movi_i64 tmp12_expr, $0x0
movi_i64 tmp12, $0x2a

call sym_add_i64, $0x5, $1, tmp2_expr,
  rax, rax_expr, tmp12, tmp12_expr
add_i64 tmp2, rax, tmp12

movi_i64 tmp12, $0x4
call sym_zext, $0x5, $1, rsi_expr, tmp2_expr, tmp12
ext32u_i64 rsi, tmp2
```

Each block of code corresponds to one of the TCG ops produced by QEMU originally; in fact, the last instruction of every block is identical with the respective original instruction. In the first block, we set the expression pertaining to the constant 42 to null (i.e., we declare the value to be concrete). In the second block, the helper `sym_add_i64` creates a symbolic expression representing the addition of two 64-bit

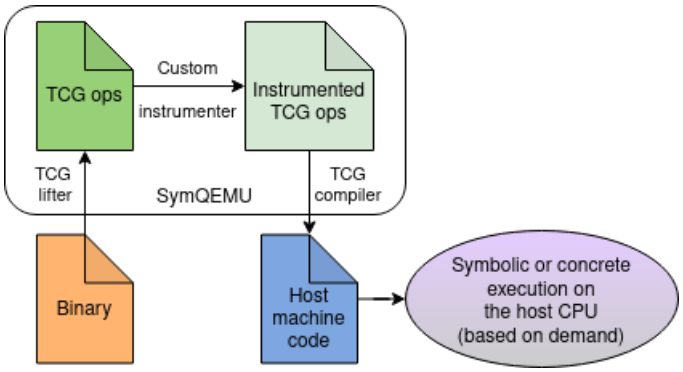

Figure 5.6: Overview of SymQEMU: the target program is translated to TCG ops as in regular QEMU (see Figure 5.5), but before the compilation to host machine code we insert instructions to perform symbolic execution at run time.

integers (using `rax_expr`, the expression corresponding to the function input). Finally, the last block calls the helper `sym_zext` with argument 4 to build an expression that translates the result of the addition to a 4-byte (i.e., 32-bit) quantity. Crucially, SymQEMU does not *perform* any of these calls to the support library at translation time (as an interpreter would)—it only emits the corresponding TCG ops and relies on the regular QEMU mechanisms to translate them to machine code. This way, symbolic formulas are constructed in native machine code without incurring the overhead associated with interpreting an intermediate language.

For the support library that constructs symbolic expressions and solves queries over them, we reuse code from SymCC, which is in turn based on QSYM (see Chapter 4). This has the advantage, in addition to saving us from having to reimplement what works well in QSYM, that it eliminates a source of noise from our evaluation: since SymQEMU and QSYM use the same logic for building up and simplifying expressions, as well as for interaction with the solver, we can be sure that observed performance differences do not originate from those orthogonal design aspects.

We currently use QEMU's Linux user-mode emulation, i.e., we emulate only the user space of the guest system. System calls are translated to fulfill the host architecture's requirements, and they are executed against the host kernel (using normal QEMU mechanics). Consequently, our symbolic analysis stops at the system-call boundary, similar to QSYM and angr. Compared to full-system emulation (as performed by S2E), this saves the effort of preparing OS images for each target architecture, and increases performance by running kernel code concretely and without emulation. Note, however, that our implementation could be extended to work with QEMU's full-system emulation if necessary (see Section 5.6).

Overall, SymQEMU adds about 2,000 lines of C code to QEMU. Furthermore, we added a few lines of C++ (less than 100) to SymCC's support library in order to support our approach to memory management (see Section 5.4.5).

5.4.3 *Platform independence*

We stated that support for multiple CPU architectures was an important goal for SymQEMU from the start. Therefore, we now examine in detail to which extent our system achieves it. (SymQEMU's claim to the second design goal, performance, is validated experimentally in Section 5.5.)

First of all, it is important to distinguish between the architecture of the computer that runs the analysis (typically called the *host*) and the architecture that the program under test is compiled for (the *guest* in QEMU parlance). Especially in firmware analysis, it is desirable for host and guest architecture to be different—the embedded device that a firmware under test runs on may lack the computing power to perform symbolic analysis at a reasonable pace, so one would typically run the symbolic executor (and, in general, any firmware tests [34]) on a more powerful machine. SymQEMU is well prepared for this use case: QEMU runs on all major host architectures.[3]

But what about guest architectures? SymQEMU leverages QEMU's TCG translators, which cover a wide range of processor types—the online documentation[4] currently lists 22 platforms including x86, ARM, MIPS and Xtensa, each comprising numerous processor types. Moreover, our modifications are almost entirely independent of the target platform: out of the 2,000 lines of C code that we added to QEMU, only 10 are specific to the guest architecture (i.e., x86 in our experiments). In particular, they perform the following tasks:

- 6 lines add space for symbolic expressions to the data structure describing the registers of the emulated CPU. Adapting them to other CPU architectures is a simple copy-paste task.
- The remaining 4 lines of code insert TCG ops on guest-level call and return instructions. This is optional, but it allows the code borrowed from QSYM to maintain a shadow call stack (see Section 5.4.7). In order to support another target architecture, one just has to identify the architecture's respective call and return primitives.

We confirmed the claim to easy adaptability by adding support for AArch64 to SymQEMU. It required 17 lines of C code, excluding the optional call and return instrumentation. Note that the current

3 Our prototype currently requires a 64-bit host system for implementation simplicity.
4 https://wiki.qemu.org/Documentation/Platforms

implementation expects 64-bit guest architectures (so that host addresses can be passed in guest registers), but there is no fundamental reason for this limitation—it could be eliminated with a one-time development effort.

In summary, SymQEMU runs on all relevant host architectures and supports the analysis of binaries compiled for any guest architecture that QEMU can handle, with negligible effort.

5.4.4 *Comparison with previous designs*

We would like to point out how SymQEMU differs from the state-of-the-art systems presented in Section 5.2.

Like angr and S2E, SymQEMU follows the traditional approach of implementing symbolic handling at the level of an intermediate representation, which significantly reduces the complexity of the implementation. However, in contrast with those two, SymQEMU performs compilation-based symbolic execution, allowing it to achieve much higher performance (see Section 5.5).

Compared with QSYM, the most important advantage of SymQEMU's design is architectural flexibility while maintaining high execution speed. Building on top of QEMU allows it to benefit from the large number of platforms that the emulator supports.

SymCC, although unable to analyze binaries, shares the compilation-based approach with SymQEMU. Both insert symbolic handling into the target program by modifying its intermediate representation, and both compile the result down to machine code that can be executed efficiently. However, SymCC is inherently designed to work in a compiler, whereas SymQEMU addresses the different set of challenges encountered in binary-only symbolic execution (see Section 2.3): where SymCC instruments LLVM bitcode during (source-based) compilation, SymQEMU instruments TCG ops during dynamic binary translation. See Section 5.4.6 for challenges that are specific to working on top of a dynamic binary translator. Moreover, SymQEMU handles mismatches between target and host architectures, an issue that does not arise in SymCC's setting because source code is mostly independent of the target architecture. In this context, we would like to emphasize that SymQEMU can support cross-architecture analysis, i.e., the CPU architecture that the program under test is compiled for does not need to match the architecture of the machine performing the analysis.

In summary, we believe that our approach combines the main advantages of angr and S2E on the one hand (i.e., platform independence) and QSYM on the other (i.e., performance), but avoids their respective disadvantages (lower performance and dependence on a particular architecture, respectively). Moreover, we found a way to apply SymCC's core idea of compilation-based symbolic execution to binaries. Tables 5.1 and 5.2 summarize the comparison.

Symbolic executor	Reference	Implementation language	Intermediate representation
angr	[74]	Python	VEX
S2E	[17]	C/C++	TCG & LLVM
QSYM	[86]	C++	none
SymCC	[63]	C++	LLVM
SymQEMU		C/C++	TCG

Table 5.1: Relevant characteristics of SymQEMU and state-of-the-art symbolic execution systems.

Symbolic executor	Speed	Multiarch	Binary-only	Cross-architecture
angr	✗	✓	✓	✓
S2E	✗	✓	✓	✓
QSYM	✓	✗	✓	✗
SymCC	✓	✓	✗	✗
SymQEMU	✓	✓	✓	✓

Table 5.2: Feature support in SymQEMU and state-of-the-art symbolic execution systems. *Speed* refers to a focus on execution speed, *multiarch* means easy portability to various guest CPU architectures, *binary-only* refers to support for analysis without source code, and *cross-architecture* means the ability to analyze programs targeting a different architecture than the host.

We now discuss some of the challenges that we faced when building SymQEMU.

5.4.5 *Memory management*

As SymQEMU executes the program under analysis, it builds up symbolic expressions that describe intermediate results and path constraints. The amount of memory required for those expressions increases over time, so SymQEMU needs a way to clean up expressions that are not needed anymore.

Before we describe SymQEMU's approach to memory management, let us discuss *why* managing memory is necessary in the first place. After all, intermediate results in any reasonable program should either have an impact on control flow or become part of the final result—in the former case, the corresponding expressions are added to the set of path constraints and thus cannot be cleaned up, and in the latter case the expressions become subexpressions in the description of the end result. So how can symbolic expressions ever become unneeded? The key insight is that *program output* is conceptually part of a program's result, but it may be produced well before the end of execution. Consider the example of an archive tool which lists the contents of an archive, printing file names one by one: after each piece of output is produced, the program can delete the associated string data, and SymQEMU should clean up the corresponding symbolic expressions. Otherwise, expressions would accumulate and, in the worst case, consume all available memory.

Ideally, we would delete symbolic expressions precisely after their last use. QSYM, whose backend we reuse, employs C++ smart pointers to this end. However, we cannot easily follow the same approach in our modified version of QEMU: TCG, the QEMU component at the center of our execution mechanism, is a dynamic translator—for performance reasons, it does not conduct any extensive analysis of translated code (unlike static compilers, which typically collect a significant amount of information related to variable scope and lifetime). This makes it difficult to efficiently determine the right place for inserting cleanup code in the translated program. Moreover, experience shows that most programs contain relatively little symbolic data and even less expressions that become garbage during execution, so we do not want our cleanup scheme to incur significant overhead in the most common case where all expressions can reside in memory until the end of program execution.

We opted for an optimistic cleanup scheme based on an *expression garbage collector:* SymQEMU keeps track of all symbolic expressions obtained from the backend, and if their number grows too large it triggers a collection. The core observation is that all live expressions can be found by scanning (1) the symbolic registers of the emulated CPU

and (2) the shadow regions in memory that store symbolic expressions corresponding to symbolic memory contents; both are known to the backend. After enumerating all live expressions, SymQEMU can compare the resulting set with the set of all expressions ever constructed, and free those that are not live anymore. In particular, when a program removes the results of a computation from registers and memory (as in the example of the archiver above), the corresponding expressions are not considered live anymore and will thus be freed. We have connected the expression garbage collector to QSYM's smart-pointer based memory management—both mechanisms need to agree that an expression is unused before it can be freed.

5.4.6 *Modifying TCG ops*

Our approach fundamentally requires the ability to insert new instructions into the list of TCG ops that represent a piece of target code. However, TCG was never meant to allow for such extensive modifications during translation—being a dynamic translator, it has a strong focus on speed. As a consequence, there is little support for programmatic editing of TCG ops. Whereas LLVM, for example, provides an extensive API for compiler passes to inspect and modify LLVM bitcode,[5] TCG simply stores instructions in a flat linked list without any navigable higher-level structure like basic blocks. Moreover, control flow is expected to be linear within a translation block (with very limited exceptions), precluding optimizations such as SymCC's embedded concreteness checks (see Section 4.3.4).

In order to minimize friction with the TCG infrastructure, our implementation emits symbolic handling for each target instruction when the instruction itself is generated. While this prevents issues with TCG's optimizer and code generator, it renders advanced static optimizations infeasible because our view is limited to only a single instruction at a time. In particular, we have very little opportunity to determine statically whether a given temporary value is concrete. Similarly, we cannot emit jumps that directly skip symbolic computations if all operands turn out to be concrete at run time. Instead, we settled on a compromise that accounts for the constraints of TCG's operating environment (in particular, the need for fast dynamic translation) while still allowing us to achieve relatively high execution speed: We perform concreteness checks in the support library—this way, we can still skip symbolic computations when the inputs are concrete, but the check costs an additional library call.

5 https://llvm.org/docs/ProgrammersManual.html#helpful-hints-for-common-operations

5.4.7 *Shadow call stack*

QSYM introduced the concept of context-sensitive basic-block pruning [86], a technique that suppresses symbolic analysis if a certain computation is encountered frequently in the same call-stack context (based on the intuition that repeating the analysis over and over in the same context will not lead to new insights). In order to support this optimization, symbolic executors need to maintain a shadow call stack, which requires keeping track of call and return instructions.

Building on top of QEMU, we faced the challenge that TCG ops are a very low-level representation of the target program. In particular, calls and returns are not represented as individual instructions in TCG but instead translate to a series of TCG ops.[6] For example, a function call on x86 results in TCG ops that push the return address onto the emulated stack, adjust the guest's stack pointer, and modify the guest's instruction pointer according to the called function. This makes it nearly impossible to recognize calls and returns reliably and in a platform-independent manner by just examining the TCG ops. We chose to optimize for robustness: in the architecture-specific QEMU code that translates machine code to TCG ops, we notify the code generator whenever a call or a return is encountered. (Hence the four architecture-specific lines of code in the x86 translator mentioned earlier—one line each for call immediate, call, return immediate, and return.) The downside is that such notifications have to be inserted into the translation code for each target architecture; however, the task is easy and the amount of code very small, so we consider it well worthwhile.

5.5 EVALUATION

In order to evaluate SymQEMU, we performed three different sets of experiments:

1. We compared it to a number of state-of-the art fuzzers with the help of Google FuzzBench.
2. Since FuzzBench does not include symbolic execution tools, we ran a comparison with popular binary-only symbolic executors on a set of real-world programs.
3. In order to assess the difference in execution speed between SymQEMU, QSYM and SymCC, we performed a benchmark comparison between those concolic executors on fixed inputs.

6 There is a *call* instruction in TCG, but it serves a different purpose.

Target	Rank		Seed corpus	Dictionary
	SymQEMU	Pure AFL		
bloaty	7	4	✓	✗
curl	5	1	✓	✗
freetype2	2	4	✓	✗
harfbuzz	2	4	✓	✗
jsoncpp	4	11	✗	✓
lcms	1	7	✗	✓
libjpeg-turbo	1	5	✓	✗
libpcap	6	10	✓	✗
libpng	1	7	✓	✗
libxml2	4	2	✓	✗
mbedtls	6	4	✓	✗
openssl	3	6	✓	✗
openthread	2	6	✓	✗
php	3	5	✓	✓
proj4	5	4	✗	✗
re2	4	6	✗	✗
sqlite3	5	2	✓	✓
systemd	3	2	✓	✗
vorbis	4	5	✓	✗
woff2	3	5	✓	✗
zlib	6	8	✓	✗

Table 5.3: Summary of the FuzzBench results for 21 targets. SymQEMU ranked first on 3 targets and outperformed pure AFL on 14.

5.5.1 *FuzzBench*

Google announced FuzzBench in March 2020 as "a fully automated, open source, free service for evaluating fuzzers".[7] It tests fuzzers in a controlled environment, comparing their performance across a large number of targets taken from Google OSS-Fuzz, a collection of fuzz targets for open-source software.[8] For each target, the service compares the edge coverage obtained by the fuzzers. Integrating a new analysis tool amounts to configuring a Docker container to set up the environment, build the target programs, and launch the analysis. We added a combination of SymQEMU and AFL to the set of analysis tools, and the FuzzBench team graciously performed a run of the experiments. In total, they ran SymQEMU and 12 fuzzer configurations on 21 targets for 24 hours, performing 15 trials per fuzzer and target (amounting to roughly 10 CPU core years).

7 https://security.googleblog.com/2020/03/fuzzbench-fuzzer-benchmarking-as-service.html

8 https://google.github.io/oss-fuzz/

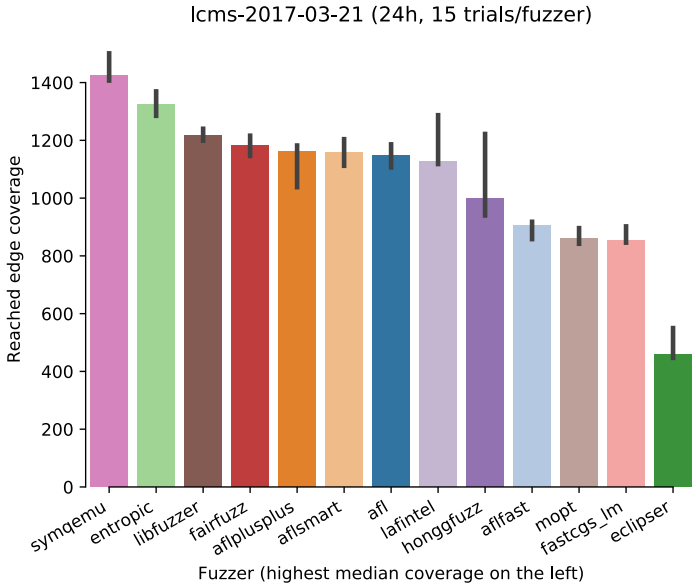

Figure 5.7: Excerpt from the FuzzBench report: Ranking by median reached coverage for the FuzzBench target *lcms*. SymQEMU outperforms all other tools on this target.

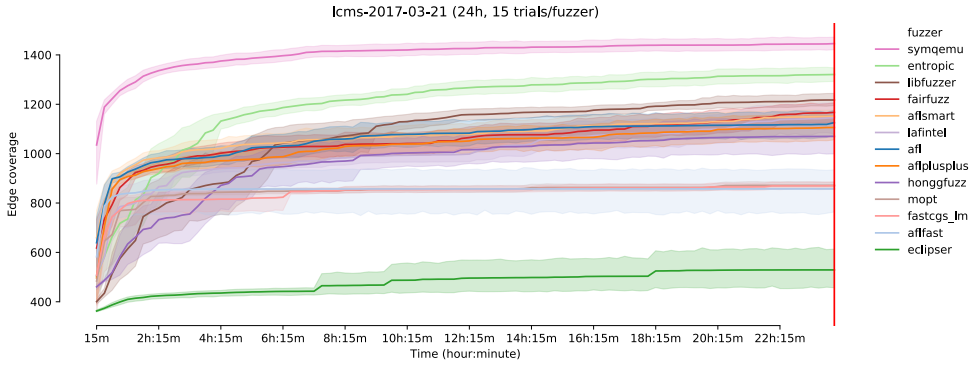

Figure 5.8: Excerpt from the FuzzBench report: Mean coverage growth over time (and 95 % confidence intervals) for the FuzzBench target *lcms*. SymQEMU outperforms all other tools on this target.

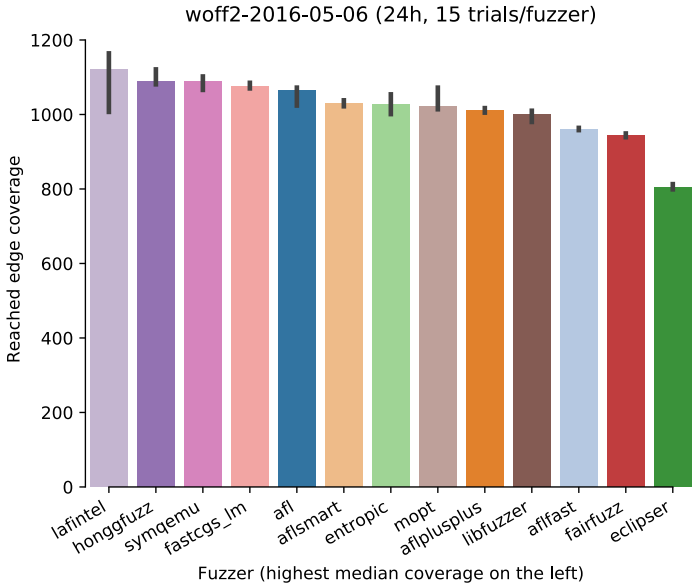

Figure 5.9: Excerpt from the FuzzBench report: Ranking by median reached coverage for the FuzzBench target *woff2*. SymQEMU reaches 3rd rank on this target.

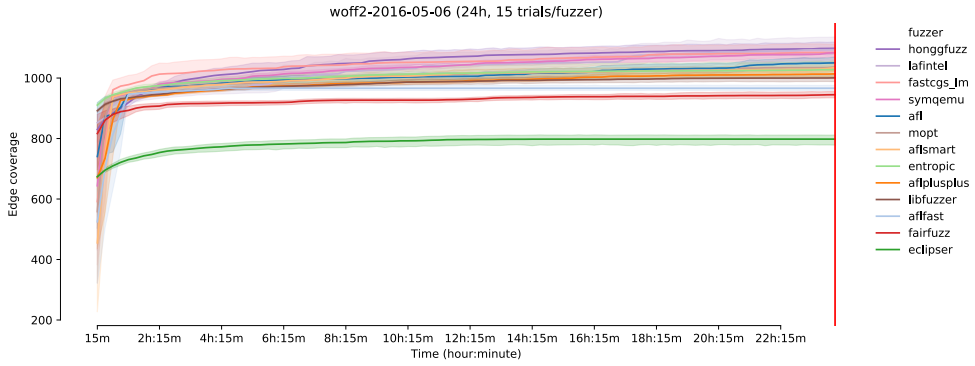

Figure 5.10: Excerpt from the FuzzBench report: Mean coverage growth over time (and 95 % confidence intervals) for the FuzzBench target *woff2*. SymQEMU reaches 3rd rank on this target.

Figures 5.7, 5.8, 5.9 and 5.10 exemplify the outcome for two targets, and Table 5.3 summarizes the results; we provide the full report online.[9] On average across all experiments, SymQEMU outperformed all fuzzers but Honggfuzz, 5 of them with statistical significance, including the popular industrial-strength tool libfuzzer. On 3 out of 21 targets, SymQEMU achieved the highest coverage among all tools, and it outperformed pure AFL on 14 targets; it is worth mentioning, however, that pure AFL consequently performed better than our hybrid fuzzer on 7 targets. The specific potential contribution of symbolic execution generally depends on several factors, including the availability of a seed corpus or dictionary, and the nature of the analyzed code—for instance, if the target makes heavy use of hash functions or other irreversible operations, the utility of symbolic execution is diminished.

Overall, we take the results as a confirmation of SymQEMU's power, especially since we have not optimized for any of the FuzzBench targets to avoid overfitting. Note also that SymQEMU achieves this without using the targets' source code, and that the overwhelming majority of the targets are accompanied by good seed corpora and/or dictionaries, where symbolic execution typically does not contribute as much in terms of raw coverage as it would if no seeds were available (see Section 5.5.2). Finally, our rather crude integration simply dedicates 50 % of CPU time to symbolic execution; we believe that a more sophisticated coordination strategy between fuzzer and symbolic executor (e.g., in the spirit of the recently presented Pangolin [38]), could further improve the results (see Chapter 6).

5.5.2 *Comparison with other symbolic execution systems*

SymQEMU's primary goal is binary-only symbolic execution. In this section, we therefore compare it to state-of-the-art tools in this space. In particular, we evaluate it against S2E because, like SymQEMU, S2E is based on QEMU (see Section 5.2.2), and against QSYM because it is the fastest binary-only symbolic executor that we are aware of (see Section 5.2.3). We omitted angr (Section 5.2.1) from the comparison because preliminary experiments showed that its execution speed is significantly lower than that of the other tools (see also Chapter 3). Finally, we added raw AFL as a baseline, and we compared with SymCC (see Chapter 4) because it introduced the concept of compilation-based symbolic execution. Note, however, that SymCC has an advantage over the other tools because it uses the source code of the program under test, i.e., it can benefit from high-level code structures and compiler optimizations. Naturally, we can only evaluate against SymCC on open-source targets.

9 http://www.s3.eurecom.fr/~seba/2020-05-24-symqemu.zip

For our comparison, we performed hybrid fuzzing of a number of target programs and measured code coverage over time. Like in the evaluation of SymCC, we used AFL's notion of coverage because it is what drives the exploration process (see Section 4.4). Following the recommendations by Klees et al. [44], we analyzed each target for 24 hours, and we repeated each experiment 30 times. In order to check for statistical significance, we used a two-tailed Mann-Whitney U test, again as recommended by Klees et al. Our targets were the open-source programs OpenJPEG, libarchive and tcpdump on the one hand, and the closed-source program rar on the other hand. The reason for choosing these programs is that (a) we used the three open-source tools for the evaluation of SymCC already (see Section 4.4), so we know that both SymCC and QSYM work on them, and (b) rar is an easy-to-obtain closed-source program whose strict requirements on the format of the input present interesting challenges to symbolic execution, and whose license does not prohibit this type of analysis. For OpenJPEG and rar, we provided a seed input of the expected format; on libarchive and tcpdump, we started with an empty corpus.

The various systems under comparison were set up as follows:

- *SymQEMU, QSYM and SymCC.* We ran those systems together with AFL, using the same integration as QSYM and SymCC (i.e., exchanging test cases via fuzzer queues in AFL's distributed mode). We executed one AFL primary instance, one AFL secondary instance, and one SymQEMU, QSYM or SymCC instance, each on one CPU core and with 2 GB of RAM. AFL was allowed to use the source code when it was available; otherwise, we ran it in QEMU mode.

- *S2E.* For S2E, we created an analysis project per target, making the test input fully symbolic when there was one, and providing a symbolic file of all zeros otherwise. We enabled the *FunctionModels* plugin and extended the *TestCaseGenerator* plugin to produce a new test case whenever a new execution state was forked.[10] We used the default searcher stack and ran the experiments in the 64-bit Debian image provided by the authors of S2E. Since S2E's parallel mode was not stable enough in our experiments, we accumulated the results from three independent analyses (to match the three CPU cores available to SymQEMU, QSYM and SymCC); see Appendix A for details. In order to assess code coverage, we evaluated the test cases with AFL after the end of the analysis.

- *Pure AFL.* We executed AFL in distributed mode, running one primary and two secondary instances, each on one CPU core with 2 GB of RAM. Like for SymQEMU, QSYM and SymCC, we

10 We contributed our extension to upstream S2E, see https://github.com/S2E/s2e/commit/8eae6e37a5e7829e77ae5cbd4fbd70656672fc46.

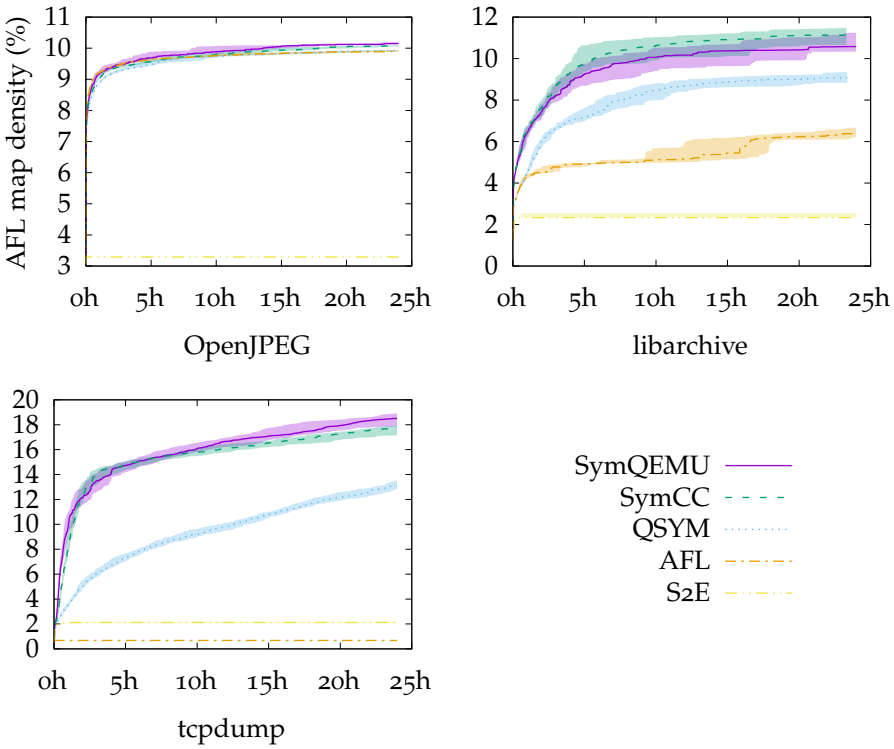

Figure 5.11: Coverage over time on the open-source targets, expressed via the density of AFL's coverage map, showing median and 95 % confidence corridor. SymQEMU achieves higher coverage than all other systems with statistical significance (Mann-Whitney U, $p < 0.005$ two-tailed), except on libarchive, where there is no statistically significant difference with SymCC. Note, however, that SymCC requires the source code of the program under test.

gave AFL access to the target's source code when it was available and used QEMU mode otherwise.

The experiments were conducted on an Intel Xeon Platinum 8260 CPU. We spent a total of roughly 5 CPU core years (4 target programs, 5 systems under comparison, 3 cores per experiment, 30 iterations, 24 hours).

Figure 5.11 shows the results for the open-source targets. We obtained coverage data for AFL and the hybrid fuzzers from the logs written by `afl-fuzz`, using the same set of AFL-instrumented binaries to evaluate each tool; for S2E, we ran the generated program inputs through `afl-showmap` (again, using the same binaries) in order to compute an equivalent coverage metric. Moreover, recall that we used identical strategies to integrate AFL with QSYM, SymCC and SymQEMU. We see that SymQEMU achieves significantly more coverage over time than both QSYM and S2E, thus outperforming those state-of-the-art binary symbolic executors. It also covers more code than pure AFL, showing the value of symbolic execution in exploring the target programs. Finally, SymQEMU somewhat surprisingly reaches a

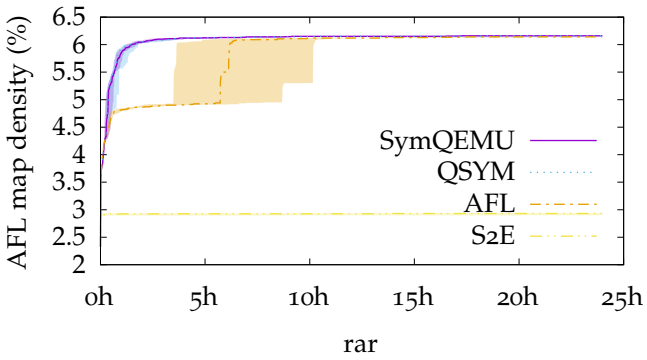

Figure 5.12: Coverage over time on the closed-source rar program, expressed via the density of AFL's coverage map, showing median and 95 % confidence corridor. All tools except S2E converge towards the same coverage level, but SymQEMU reaches it faster than AFL and therefore requires less computing power per coverage. Moreover, its speed is similar to QSYM's, but QSYM cannot be easily ported from x86 to other targets.

coverage level that is comparable with SymCC's results, even though SymCC has access to the targets' source code and therefore more potential for optimization. Manual investigation shows that SymCC does not use this potential to the maximum extent possible; for example, it does not trigger another memory-to-register optimization pass after inserting its instrumentation (resulting in unnecessary memory operations in the target program), nor does it use link-time optimization to inline calls to the support library. We believe that this is the main reason why a binary-only symbolic executor like SymQEMU can keep up with a source-based tool like SymCC. In summary, the results confirm that SymQEMU is more efficient than the other binary-only symbolic execution systems in our comparison.

In our analysis of libarchive, SymQEMU found an input that leads to a use-after-free error on the heap. The bug can be triggered, for example, by making a user list the contents of a manipulated archive with the *bsdtar* utility, and we consider it likely to be exploitable. We have reported the issue to the developers of libarchive; at the time of writing, we have not received a reply.

Figure 5.12 displays the results for the closed-source rar program. SymQEMU, QSYM and AFL all converge towards the same level of coverage, but SymQEMU reaches saturation as fast as the less architecturally flexible QSYM and faster than AFL. Note further that SymQEMU and QSYM quickly discover paths that pure AFL (i.e., without symbolic execution) needs more time to find. S2E cannot analyze as much code as the other tools but arguably covers more of the data space on the discovered paths.[11] This experiment shows that

11 https://ccadar.blogspot.com/2020/07/measuring-coverage-achieved-by-symbolic.html

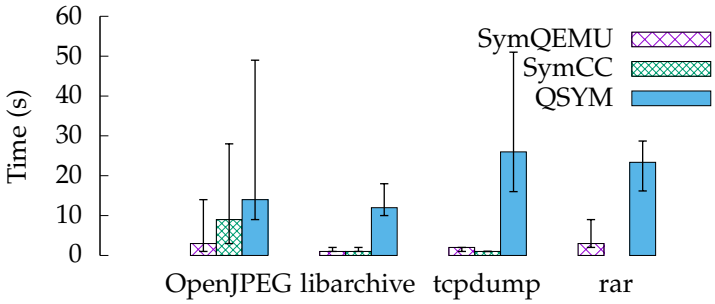

Figure 5.13: Target execution times per symbolic executor and target program. Note that SymQEMU is faster than QSYM and at least as fast as the source-based SymCC. The notion of execution time is not applicable to S2E; SymCC cannot analyze rar because the source code is not available.

SymQEMU can work with closed-source targets, like other binary-only symbolic executors, but with the additional advantage of easily supporting a large number of target architectures.

It is interesting to note that symbolic execution generally contributes the most in terms of code coverage when no seed inputs are available, as demonstrated by our analysis of libarchive and tcpdump. On OpenJPEG and rar, in contrast, the seed files give AFL sufficient information to also achieve a good coverage level in relatively little time.

Finally, Figure 5.13 shows the execution times of the symbolic execution engines in our experiments, providing evidence that SymQEMU is consistently faster than QSYM and at least on par with the source-based SymCC. We omitted S2E from the figure because there is no equivalent notion of execution time for its approach: while we could measure how long each execution state exists, this would ignore the fact that S2E performs many checks that the other systems delegate to fuzzer and sanitizers, and hence would put S2E at an unfair disadvantage in the comparison.

5.5.3 *Benchmark comparison*

We have seen that SymQEMU outperforms state-of-the-art binary-only symbolic executors in real-world hybrid fuzzing. Let us now check our hypothesis that those results are indeed due to SymQEMU's high execution speed. To this end, we performed a third set of experiments with the goal of assessing precisely how fast SymQEMU executes code in comparison with the other two concolic executors in our evaluation, SymCC and QSYM.

The core idea of this experiment is to run concolic execution on a fixed set of inputs, therefore making all systems in the comparison follow the same paths on the same target programs. In other words, we remove one variable from the comparison: the choice of paths

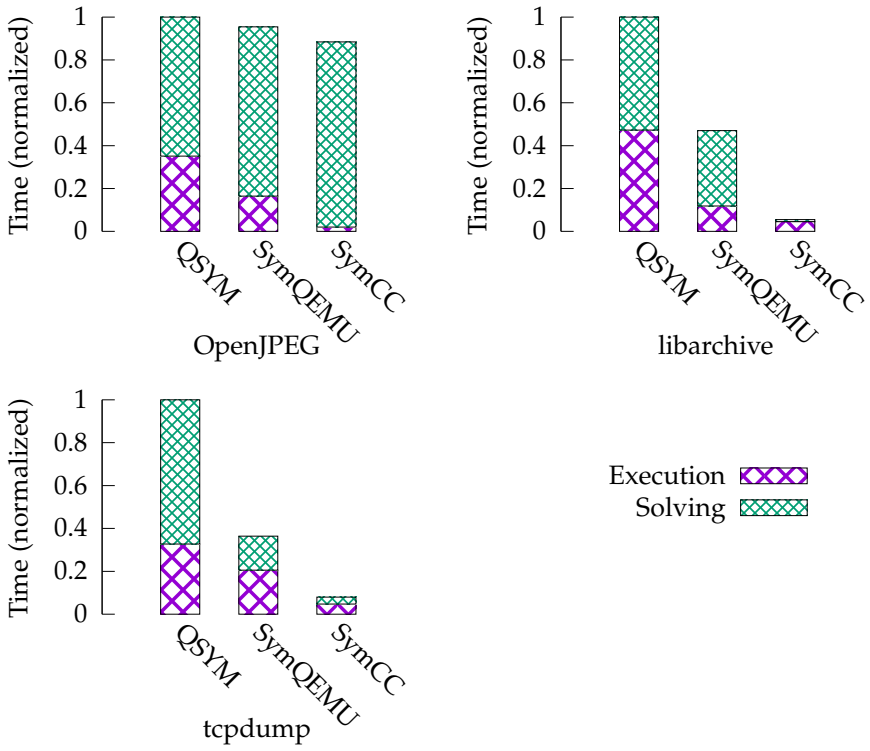

Figure 5.14: Time spent in execution and SMT solving, respectively, averaged across concolic execution of a fixed set of test cases (1000 cases per target, chosen at random and analyzed in each of the three symbolic executors). Times are normalized to the total execution time of the slowest engine per target to show the differences in the overall amount of time required to complete the benchmark.

to follow. Concretely, for each of the three open-source targets used in Section 5.5.2 we combined the test cases found by SymQEMU, SymCC and QSYM during the 24-hour hybrid-fuzzing session; then we selected 1000 test cases per target at random. We performed concolic execution on the selected inputs, measuring the time spent in execution and SMT solving, respectively.

Figure 5.14 and Table 5.4 show the observed time split per target and symbolic executor. We see that, on all three targets, SymQEMU spends less time in execution than QSYM; this provides evidence that SymQEMU's higher performance in hybrid fuzzing (see Figure 5.11) is indeed due to higher execution speed. The source-based SymCC spends even less time in execution than both SymQEMU and QSYM because, unlike the binary-only symbolic executors, SymCC does not incur the overhead of dynamic binary translation or dynamic binary instrumentation.

It is also interesting to note that the three systems invest different amounts of time in SMT solving. Since the program paths are fixed and the symbolic executors use the same backend to interact with the solver, we conclude that there is a difference in the difficulty of

		QSYM		SymQEMU		SymCC	
OpenJPEG	Exec	8.5 h	35.1 %	4.0 h	17.2 %	0.5 h	2.3 %
	SMT	15.8 h	64.9 %	19.2 h	82.8 %	21.0 h	97.7 %
	Total	24.3 h		23.2 h		21.5 h	
libarchive	Exec	6.8 h	47.3 %	1.7 h	25.1 %	0.7 h	82.6 %
	SMT	7.6 h	52.7 %	5.0 h	74.9 %	0.1 h	17.4 %
	Total	14.3 h		6.7 h		0.8 h	
tcpdump	Exec	8.2 h	32.7 %	5.2 h	56.5 %	1.2 h	59.3 %
	SMT	16.9 h	67.3 %	4.0 h	43.5 %	0.8 h	40.7 %
	Total	25.1 h		9.1 h		2.0 h	

Table 5.4: Results of our benchmark comparison on fixed inputs, visualized in Figure 5.14.

SMT queries. Manual inspection confirms that SymCC's queries are shorter and less nested than those generated by the other systems (except on the OpenJPEG target, where we see a lot of arithmetic and bit-level operations in all systems' queries, which we attribute to the compression algorithm of the JPEG format). The difference in difficulty is likely due to the different intermediate representations that the analyses are based on. In particular, our observation that SymCC often generates simpler queries and consequently spends less time in the SMT solver than the other two systems provides evidence for our earlier hypothesis that high-level intermediate representations lead to simpler SMT queries (see Section 3.5).

In summary, we have shown that SymQEMU outperforms state-of-the-art binary-only symbolic executors in real-world hybrid fuzzing, and that the reason for its higher performance is its fast execution component. In comparison with QSYM, SymQEMU achieves 19 % higher coverage on average after 24 hours (Figures 5.11 and 5.12, geometric mean) and 58 % faster execution in the benchmark experiment (Figure 5.14, geometric mean).

5.6 FUTURE WORK

We have several ideas for future work based on SymQEMU, which we document in this section.

5.6.1 *Full-system emulation*

SymQEMU currently performs symbolic execution of Linux user-mode binaries. It would be interesting to extend it to full-system analysis. Especially in the embedded space, it is common for firmware to run on custom operating systems or even directly on hardware [55]; analyzing such programs would require support for full-system emulation.

We believe that it is possible to implement such a system on top of SymQEMU. The basic process of lifting the target to TCG ops, instrumenting those, and compiling the result down to host machine code would stay the same. One would have to add a mechanism to introduce symbolic data into the guest system (e.g., inspired by S2E's fake-instruction technique), and the shadow-memory system would have to account for the virtual MMU when mapping between guest memory and symbolic expressions. The result would be a symbolic executor that could reason about kernel code in addition to user-space programs. Moreover, the extended system would be able to analyze code for non-Linux operating systems, as well as bare-metal firmware.

5.6.2 *Caching across executions*

Hybrid fuzzing is characterized by a large number of successive executions of the same program. Being a dynamic translator, QEMU (and hence, SymQEMU) translates the target program on demand, at run time. And although the results of the translation are cached for the duration of a single execution, they are discarded when the target program terminates. We conjecture that the overall performance of hybrid fuzzing with SymQEMU could be improved by caching translation results *across executions*. The main challenges would be to ensure that the target is loaded deterministically, and special handling would need to be put in place for self-modifying code. Therefore, the potential benefit of this optimization depends heavily on the characteristics of the program under test.

5.6.3 *Symbolic QEMU helpers*

QEMU represents target machine code with TCG ops. However, some target instructions are too complex to be efficiently expressed in TCG, especially on CISC architectures (e.g., Intel's SSE extensions). In such cases, QEMU uses *helpers:* built-in compiled functions that can be called from TCG, emulating single complex instructions of the target architecture. Since helpers operate outside the regular TCG framework, SymQEMU's instrumentation at the TCG level cannot insert symbolic handling into them. The result is implicit concretization, yielding a loss of precision in the analysis of targets that make heavy use of complex instructions.

We see two ways to implement symbolic handling of QEMU helpers when the need arises:

1. One approach is to hand-craft symbolic equivalents for each required helper, much like the function summaries used for common libc functions in some symbolic executors.[12] This approach

12 E.g., http://s2e.systems/docs/Plugins/Linux/FunctionModels.html.

is easy to implement but does not scale to large numbers of helpers.

2. An alternative is to build symbolic versions of the helpers automatically. To this end, SymCC could be used to compile symbolic tracing into the helpers, whose source code is available as part of QEMU. The resulting binaries would be compatible with SymQEMU because SymCC uses the same backend for symbolic reasoning. S2E follows a similar approach when compiling the helpers to LLVM bitcode for interpretation in KLEE.

Since such improvements would provide benefit mainly for very specific targets that make heavy use of complex instructions, we leave them to future work.

5.7 RELATED WORK

We now place SymQEMU in the context of previous work.

5.7.1 *Binary-only symbolic execution*

Angr [74], S2E [17] and QSYM [86] have been described in Section 5.2, and we have compared them to SymQEMU in Section 5.4.4. Mayhem [15] is a high-performance interpreter-based implementation of symbolic execution that won the DARPA CGC competition; unfortunately, it is not publicly available for comparison. Triton [69] has a symbolic execution component that can operate in two different modes: one uses binary translation (like QSYM), the other works with CPU emulation (like S2E and angr). Eclipser [18] covers some middle ground between fuzzing and symbolic execution by assuming linear relations between branch conditions and input data; the constraint simplification increases the system's performance at the cost of reasoning power, so that Eclipser cannot find all the paths that conventional symbolic execution can. In a similar vein, Redqueen [2] searches for correspondence between branch conditions and input bytes using a number of heuristics. SymQEMU, in contrast, implements "full" symbolic execution.

5.7.2 *Run-time bug detection*

Hybrid fuzzing relies on the fuzzer and sanitizers to detect bugs. Address sanitizer [72] is a very popular sanitizer that checks for certain memory errors. Since it requires source code to produce instrumented target programs, Fioraldi et al. have recently proposed QASan [27], a QEMU-based system that implements similar checks for binaries. There is a plethora of other sanitizers, often requiring source code [75]. We conjecture that it would be possible to use many of them on

binaries via emulation in the spirit of QASan. They could complement hybrid fuzzing with SymQEMU, but such work is orthogonal to what we present here.

5.7.3 *Hybrid fuzzing*

Driller [77] is a hybrid fuzzer based on angr, similar in concept to QSYM but slower because of its Python implementation and interpreter-based approach [86]. In comparison with QSYM and SymQEMU, it uses a more elaborate strategy to coordinate fuzzer and symbolic executor: it monitors the fuzzer's progress and switches to symbolic execution whenever the fuzzer appears to encounter obstacles that it cannot overcome on its own. In a similar spirit, the recently proposed Pangolin [38] enhances the fuzzer's benefit from symbolic execution by providing the fuzzer not only with new test cases but also with an abstraction of the symbolic constraints, along with a fast sampling method; using those, the fuzzer can generate new inputs that have a high probability of fulfilling the path constraints determined by symbolic execution.

We believe that more sophisticated coordination strategies between fuzzer and symbolic executor can greatly enhance the performance of hybrid fuzzing (see also Chapter 6). However, since such improvements are orthogonal to the speed of the symbolic executor (which is our main concern), they are outside the scope of this thesis.

5.8 CONCLUSION

We have presented SymQEMU, a novel approach to apply compilation-based symbolic execution to binaries. Our evaluation shows that SymQEMU significantly outperforms state-of-the-art binary symbolic executors and even keeps up with source-based techniques. Moreover, SymQEMU is easy to extend to many target architectures, requiring just a handful lines of code to support any architecture that QEMU can handle. Finally, we have demonstrated SymQEMU's real-world use by discovering a previously unknown memory error in the heavily tested libarchive library.

AVAILABILITY

We will make all source code for SymQEMU publicly available at a later point in time. We will also provide detailed instructions to reproduce our experiments, and we will share the raw results of our own evaluation.

6 CONCLUSION AND FUTURE WORK

Throughout this thesis, our goal has been to improve the performance of symbolic execution by increasing its speed. We have presented compilation-based symbolic execution, a novel approach inspired by a systematic observation of prior work. We have shown that our compiler-based technique outperforms existing alternative approaches with a considerable margin, and finally we have extended the reach of compilation-based symbolic execution—originally a source-based technique—to the realm of binary-only analysis. It is my hope that this work provides value to the community in its quest to construct more and more practical symbolic execution systems.

However, increasing the speed of symbolic execution is only one way to improve its overall performance, and there are other options besides accelerating the execution component, which has been the aim of my work. The remainder of this chapter provides an outlook on avenues of future research that I consider promising, complementing and improving on my own contribution.

Our observations from previous chapters suggest that the cost of symbolic execution is dominated by two core factors: execution—i.e., running programs in a monitored environment in order to trace computations symbolically—and solving. This notion is also supported by measurements from Yun et al. [86]. Furthermore, there is a coordination cost: standalone symbolic executors need to fork execution states and/or snapshot execution, and systems used in hybrid fuzzing need to communicate with a fuzzer. Finally, efficiency in terms of bugs discovered per time is heavily impacted by the exploration strategy: which parts of a program are the most interesting to examine? The following sections discuss each of those aspects in more detail.

6.1 EXECUTION

Increasing the performance of the execution component in symbolic executors has been the primary goal of the work described in this thesis. I am convinced that compiler-inserted instrumentation has great potential; Chapters 4 and 5 have already suggested various directions of future work in this area.

Moreover, it could be beneficial to improve the abstract understanding that symbolic executors have of the target code. Consider,

for example, the following code snippet, based on a user report for SymCC:[1]

```
const unsigned expected = 42;
unsigned input; // symbolic user input

// ...

unsigned result = 0;
for (unsigned i = 0; i < input; i++) {
  result++;
}

printf("%s\n", (result == expected) ? "yes" : "no");
```

The program reads an integer from the user, then increments the variable `result` in a loop until it equals `input`. Finally, it compares `result` to the constant `expected`. As humans, we can recognize that the loop is equivalent to the assignment `result = input`, and therefore the required input to make the last line print "yes" is the value of the constant `expected`.[2] Symbolic execution systems, however, currently lack such abstraction capabilities; they analyze the program instruction by instruction, producing new test cases (or forking execution states) when encountering the comparison `i < input`. Eventually, they find the solution, but their approach is roughly equivalent to trying each possible value for `input`.

While it is theoretically impossible to mechanically "understand" every possible fragment of code, we already have good heuristics for many common cases. Modern compilers, for example, can statically simplify the loop in the above example to `result = input`; after this optimization, symbolic execution can immediately infer the expected input in the last line. It would be interesting to explore to which extent the "code understanding" of compilers and static analyzers can be incorporated into and benefit symbolic execution systems. Kapus et al. have recently presented work on symbolically summarizing a special type of loops that they call *memoryless loops* [40]. State merging [46] produces a type of loop summary dynamically but faces its own set of challenges.

6.2 SOLVING

SMT solving is a major source of power for symbolic execution. It would be hard to imagine a symbolic executor that does not rely on the advanced reasoning capabilities of modern SMT solvers. However, answering an SMT query is a difficult problem, with the exact meaning

1 https://github.com/eurecom-s3/symcc/issues/14

2 I found it highly instructive to replace one or more of the types in the example with `int` or `uint64_t`: between integer overflows, implicit conversions and undefined behavior, I was almost unable to work out the program's behavior.

of "difficult" depending on the logic underlying the query. The queries that arise in symbolic execution typically use at least the logic of quantifier-free formulas over bit vectors, commonly referred to as `QF_BV` in the SMT community. Many solvers handle this logic by translating queries into instances of the Boolean satisfiability problem (SAT) in a process called "bit blasting", and SAT is well known to be NP-complete. It is therefore reasonable to expect that no solver can efficiently answer every query, and that we need good heuristics adapted to our use case.

Intuitively, representing the path constraints of symbolic execution as SAT instances discards a lot of information about structure: variables in a program are typically related via arithmetic operations, often even in linear relations. Such high-level structure may be exploitable in the solver. (Admittedly, bit-level operations and integer overflows complicate the picture.) For example, STP [30], the default solver in KLEE [12], was specifically developed for software analysis: it features a preprocessing step that efficiently solves linear arithmetic problems before bit blasting; Kapus et al. have recently implemented a similar approach for KLEE's Z3 backend [41]. Similarly, a specialized solver could represent and reason about operations on common data structures like lists, sets and dictionaries. However, most recent symbolic executors use Z3 [23], a general-purpose SMT solver from Microsoft Research. While there are good reasons to rely on a solver whose development is supported by a large corporation, the community may miss some targeted optimizations that could be applied to the workload of symbolic execution.

Another aspect is the relation between queries: a concolic executor (or a single execution state of a forking symbolic executor) collects path constraints as it executes the program, effectively building up a large conjunction of terms that is only ever added to. SMT solvers can exploit this structure by reusing the results of earlier queries. angr [74], for example, uses the solver in this iterative mode, and KLEE preprocesses queries based on subset and superset relations with earlier queries in order to reduce the load on the SMT solver. However, research on the SMT side with a focus on the particular sequences of SMT queries generated by symbolic execution might lead to further improvements.

6.3 COORDINATION

Symbolic execution does not typically run in a vacuum. There are other tools and processes to coordinate with, and the coordination strategy has an impact on the overall system's performance. Here we focus on the common case of hybrid fuzzing, where a symbolic executor cooperates with a fuzzer to discover software defects.

The core idea of hybrid fuzzing is to combine the efficiency of fuzzers with the sophisticated reasoning capabilities of symbolic execution: the fuzzer quickly explores the easy-to-reach parts of the target program, while symbolic execution generates test cases to explore the more difficult-to-reach areas. In the simplest case, fuzzer and symbolic executor just exchange test cases. This is the approach followed by QSYM [86], SymCC (Chapter 4) and SymQEMU (Chapter 5); it is easy to implement and flexible, yet it wastes a lot of potential. For example, as long as the fuzzer makes progress, CPU time is probably best invested into fuzzing, while symbolic execution should be used only when complicated constraints need to be solved; Driller [77] implements a heuristic along those lines, but more sophisticated scheduling strategies are conceivable. Moreover, a symbolic executor that just passes test cases to the fuzzer effectively throws away a lot of its results: the path constraints, for example, can be used to guide the fuzzer's mutation of the new test cases. Pangolin [38] implements this idea with great success, but again, more research on the topic may lead to even more efficient techniques. Finally, there are mundane technical optimizations that could be worked on: file-based coordination strategies, for example, may become a bottleneck in case of I/O saturation, and symbolic executors need clear indications which test cases the fuzzer considers most promising.

6.4 EXPLORATION

Standalone symbolic executors typically feature a component that assigns priorities to execution states and schedules them for exploration. In hybrid fuzzing, concolic executors rely on the fuzzer for similar tasks. In either scenario, current approaches largely base their decisions on code coverage. Additionally, the community is experimenting with directed approaches, where test cases are judged based on their estimated distance to some point of interest in the target program [7, 16]; KATCH [54] selects such points of interest based on changes in the version-control history, and ParmeSan [58] uses potentially vulnerable code patterns identified by sanitizers. However, even with directed strategies, success is often defined in terms of reaching a particular piece of code, disregarding the question of coverage in the *data space*. In the most basic example of a pointer dereference, merely reaching the dereferencing instruction does not reveal a bug—unless the pointer happens to have an invalid value. Ankou [53] has recently proposed a more elaborate function for judging the merit of individual test cases, which accounts for hit counts in addition to regular branch coverage. Research into functions that take data-space coverage into account may help to direct symbolic execution towards parts of a program that are most likely to contain defects.

This concludes the outlook on future work, and my thesis. I hope the reader enjoyed reading it as much as I took pleasure in its creation.

Sebastian Poeplau,
August 2020

A

S2E RESOURCE CONSUMPTION

We encountered a few challenges related to resource consumption when setting up S2E for comparison with SymQEMU (see Chapter 5). While they are not essential to the discussion, we still think they are interesting to document.

A.1 PARALLEL S2E

S2E has a parallel mode, in which it starts multiple processes and assigns each process a dedicated portion of the state tree.[1] Initially, we tried to use this mode to compensate for the fact that the other symbolic executors in our comparison each use 3 CPU cores. However, in our setup, parallel mode was prone to deadlocks and crashes that turned out to be hard to debug and fix. As a workaround, we started 3 independent S2E instances, relying on randomization of the search strategy to prevent them from exploring the same paths. This is not ideal but seemed the fairest approach given the circumstances.

A.2 MEMORY LIMITS

Like the other systems in the comparison, we attempted to execute S2E with 2 GB of RAM per CPU core (i.e., per S2E process). Setting a hard limit via cgroups, as we did for the other systems, turned out impossible because S2E runs the entire analysis in a single long-running process—if the operating system terminates that process due to excessive memory consumption, the analysis fails. (In contrast, AFL, SymQEMU, QSYM and SymCC create many short-lived analysis processes; if one of them fails, the analysis just continues with the next one.)

S2E provides the *ResourceMonitor* plugin for such cases.[2] Its task is to monitor memory consumption (with the limit defined via a cgroup) and prevent further forking or terminate execution states as consumption approaches the limit. Unfortunately, in our experiments, the plugin did not reduce memory consumption aggressively enough—while the analysis ran slightly longer, it would still eventually exceed the memory limit and trigger the operating system's OOM killer. We experimented with adjusting the plugin (e.g., trigger earlier than the default threshold of 95 % memory consumption) but could not find a configuration that would permit the analysis to run for 24 hours.

1 http://s2e.systems/docs/Howtos/Parallel.html
2 http://s2e.systems/docs/FAQ.html#how-to-keep-memory-usage-low

Finally, we resorted to the following strategy: instead of enforcing 2 GB per S2E instance, we only imposed a total limit on the cumulative memory consumption of all S2E processes. As a result, some processes were terminated by the operating system whereas others were allowed to consume significantly more than 2 GB of RAM and thus analyze the target for 24 hours. The reason that this strategy did not result in higher variance of the results for S2E (see Figure 5.11) is that most execution states were forked in the first few minutes of the analysis, i.e., before any process hit the memory limit.

B

FRENCH SUMMARY OF THE THESIS

Ce chapitre est un résumé de la thèse en français.

B.1 INTRODUCTION

L'exécution symbolique est un technique pour l'analyse systématique et automatique de logiciels. Elle a été inventée par James C. King en 1976 [43] pour simplifier le test des logiciels, ce qui était surtout un travail manuel. L'idée est de tracer l'exécution du logiciel pour comprendre exactement comment chaque résultat intermédiaire est calculé. À chaque branchement conditionnel le système connaît précisément l'expression qui est évaluée et comment elle est dérivée de l'entrée utilisateur. Un solveur peut ensuite déterminer une nouvelle entrée qui changera le résultat du branchement. Cette stratégie peut simplifier la création d'une entrée qui mène à un point d'intérêt dans le logiciel analysé, et la répétition automatique aide à l'exploration de tous les chemin d'exécution possibles.

La réalisation de King était très interactive : elle demandait une décision d'un utilisateur humain à chaque branchement. En outre, comme les performances des ordinateurs de l'époque étaient faibles en comparaison avec ceux que nous avons aujourd'hui, le solveur de King n'était pas très puissant. Les réalisations récentes sont plus automatisées, et elles exploitent les avancements importants dans le domaine des solveurs SAT (satisfiabilité booléenne) et SMT (satisfiabilité modulo des théories) des années 90 et 2000. Les systèmes modernes d'exécution symbolique sont plutôt automatique, et leur solveurs sont très capables.

Malgré les importants progrès, l'exécution symbolique souffre toujours de problèmes de performance pour lesquelles il y a deux raisons principales :

1. Les problèmes SMT sont, en général, NP-complets. Bien qu'il y ait des approches heuristiques qui fonctionnent en pratique, il n'est probablement pas possible de trouver une solution en temps polynomial (sauf si $P = NP$). D'ailleurs, il y a des logiciels qui effectuent des calculs difficilement réversibles, comme le chiffrement. Mais la réversibilité de toute opération dans l'exécution d'un logiciel est nécessaire pour l'exécution symbolique.

2. L'exécution d'un logiciel sous surveillance afin de tracer toute opération est coûteuse en temps de calcul. La plupart des réalisations actuelles fonctionnent comme un interpréteur d'un langage

de programmation ou d'une représentation intermédiaire, ce qui est plus coûteux que l'exécution normale dans le processeur.

Ma thèse se focalise sur le deuxième aspect, qui sera décrit plus en détail dans les sections suivantes. (Un autre problème de l'exécution symbolique et ce qui s'appelle « path explosion » en anglais – la croissance exponentielle de la nombre de chemins possibles. Mais comme c'est une difficulté de toute stratégie qui essaie de traverser un logiciel par tous les chemins possibles, on va l'ignorer dans le contexte de ce résumé.)

Des travaux récents ont démontré que la combinaison de l'exécution symbolique avec le fuzzing est prometteur. Le fuzzing est une technique plus simple pour tester des logiciels : le programme analysé est exécuté avec des entrées aléatoires, en espérant qu'une d'entre elles trouve un dysfonctionnement dans le logiciel. Bien que c'est un mécanisme simple et rapide à implémenter, le fuzzing ne trouve souvent pas tous les chemins dans un logiciel parce qu'il manque un moyen pour résoudre les questions difficiles associées à la création de nouvelles entrées pour certains chemins (le rôle du solveur SMT dans l'exécution symbolique). La combinaison du fuzzing avec l'exécution symbolique est intéressant parce qu'elle permet de découvrir vite une large partie du logiciel par le fuzzing alors que les chemins plus difficiles à trouver sont découverts par l'exécution symbolique.

Dans le contexte de ma thèse, j'ai travaillé sur des méthodes pour accélérer l'exécution symbolique afin d'atteindre une performance suffisante pour appliquer ce type d'analyse a des situations réelles, toujours en combinaison avec le fuzzing. Dans la section B.2, je décrie l'analyse de systèmes d'exécution symboliques actuelles, en particulier l'impact de la représentation intermédiaire sur la performance. La section B.3 présente SymCC, une réalisation inspirée par les résultats de cet analyse. SymCC permet l'exécution symbolique très efficace de logiciels dont le code source est disponible. Enfin, la section B.4 résume le travail sur SymQEMU, un système complémentaire pour l'exécution symbolique de fichiers binaires. Le principe clé de SymCC et de SymQEMU est d'intégrer des capacités d'analyse directement dans un binaire, ainsi il n'est pas nécessaire d'interpréter le programme analysé.

B.2 ANALYSE DE L'EFFET DE LA REPRÉSENTATION INTERMÉDIAIRE

Cette section (et le chapitre correspondant de la thèse) est basée sur l'article « Systematic Comparison of Symbolic Execution Systems : Intermediate Representation and its Generation », publié lors de l'Annual Computer Security Applications Conference (ACSAC 2019) à San Juan, Porto Rico.

En règle générale, les moteurs d'exécution symbolique traduisent le programme testé en un *représentation intermédiaire* (RI) qu'ils peuvent ensuite exécuter symboliquement. La génération de la RI à partir du binaire peut être la seule solution lorsque le code source n'est pas disponible. Tester directement le binaire a également l'avantage de tester le produit mis en production, indépendamment du langage source et du compilateur. Cependant, lorsque le code source est disponible, les deux approches sont possibles et le choix de la façon de générer la RI est un facteur distinctif entre les implémentations différentes. Il y a une certaine sagesse conventionnelle qui l'entoure : l'intuition est que la sémantique du code source de haut niveau (par exemple, les limites des tableaux mémoire, types) peuvent être préservés et rendront l'exécution symbolique et la recherche de dysfonctionnements plus efficace. Cependant, aucune étude systématique ne soutenait de telles affirmations. Nous avons donc systématiquement évalué la manière dont le choix des RI et le processus de les générer influencent de divers aspects de l'exécution symbolique.

Nous avons sélectionné plusieurs implémentations populaires, chacune avec son propre mécanisme de génération de RI, et nous avons comparé leur performances. En particulier, nous avons répondu aux questions de recherche suivantes :

1. Y a-t-il un avantage à générer la RI à partir du code source par rapport aux RI générées à partir de binaires ?

2. Existe-t-il des différences significatives entre l'exécution symbolique de différentes RI généré à partir des mêmes programmes ? Qu'en est-il du cas particulier d'exécuter symboliquement le code machine directement ?

En cours de route, nous avons découvert que la tâche d'ingénierie vraisemblablement simple de mettre en place un certain nombre de moteurs d'exécution symbolique dans un environnement stable afin d'en faire un comparaison équitable est en réalité un réel défi. Cela a été une inspiration majeure dans notre quête pour rendre nos propres systèmes faciles à utiliser et comparer.

B.2.1 *Représentation Intermédiaire*

Lors de l'émulation de l'exécution d'un programme, l'exécution symbolique fait face au défi que les jeux d'instructions des processeurs modernes sont complexes et constitués d'un grand nombre d'instructions ; leur écrire un émulateur symbolique n'est pas anodin. Par conséquent, il est courant de transformer le programme testé dans une représentation intermédiaire, qui est ensuite émulé. L'exécution symbolique au niveau RI augmente également la portabilité : pour prendre en charge une nouvelle architecture, il suffit de réimplémenter le traducteur RI, tandis que le reste du système peut rester inchangé.

Les moteurs d'exécution symboliques diffèrent dans l'approche choisie pour la traduction en RI que ce soit à partir d'un binaire ou d'un code source. Nous référons au processus comme *génération RI*, que l'artefact initial soit un fichier binaire de code machine ou code source, car le terme lifting (« levage ») qui est utilisé en anglais convient uniquement pour la génération à partiir du code machine. Le choix du mécanisme de génération RI a une influence considérable sur plusieurs aspects de l'exécution symbolique, ce qui était la motivation de cette étude.

B.2.2 *Les solveurs SMT*

Les moteurs d'exécution symbolique doivent résoudre les contraintes de chemin pour les valeurs d'entrée ; en d'autres termes, ils doivent résoudre des formules dans la logique des vecteurs de bits et des tableaux. Le domaine de la résolution SMT fournit des outils pour résoudre ce problème (généralement difficile [45]) : dans de nombreux cas, les solveurs SMT modernes peuvent résoudre ces requêtes difficiles dans temps acceptable, en utilisant diverses heuristiques qui sont elles-mêmes un domaine actif de recherche. C'est cependant dans le meilleur intérêt de tout moteur d'exécution symbolique de générer des requêtes sous une forme que les solveurs SMT peuvent résoudre rapidement. Nous conjecturions que la façon dont la RI est généré a un impact profond sur la complexité des requêtes SMT résultantes.

B.2.3 *Les approches analysées*

Dans cette étude, nous avons comparé des approches de génération RI classiques, chacune représentée par un outil. Notre ensemble de test comprenait KLEE [12] pour la génération RI basée sur le source, S2E [17] pour la génération binaire, angr [74] comme une approche binaire avec une autre RI et QSYM [86] comme représentant pour les systèmes qui n'utilisent pas de RI du tout. Le cas échéant, nous avons ajouté les résultats de McSema [24], un traducteur statique du code machine à la RI LLVM basé sur le désassembleur commercial IDA Pro.

B.2.4 *Évaluation*

Afin de répondre à nos questions de haut niveau, nous devions choisir des propriétés concrètement mesurables qui fourniraient les preuves nécessaires. Qu'attendons-nous d'une technique de génération RI idéale pour l'exécution symbolique ? Puisque nous allons exécuter la RI, nous voulons qu'elle soit facile à interpréter efficacement, et nous la voulons être concis. De plus, étant donné que la résolution SMT consomme une partie considérable du temps d'analyse global, nous aimerions que la RI conduise à des requêtes SMT auxquelles

le solveur peut répondre rapidement. Nous avons donc évalué les mécanismes de génération RI différents sous trois aspects motivés par les observations ci-dessus :

1. Dans quelle mesure la traduction en RI augmente-t-elle ou diminue-t-elle le nombre d'instructions ?
2. Dans quelle mesure pouvons-nous exécuter efficacement la RI résultante ?
3. Quelle est la difficulté des requêtes SMT dérivées de la RI ?

Taille du code

Nous avons déjà mentionné l'intuition que la RI dérivée du code source contient plus d'informations de haut niveau que la RI binaire ; une manière plus précise d'exprimer cette intuition est de dire que nous nous attendons à ce que la RI dérivée du source contienne plus d'informations sémantiques par instruction RI que la RI dérivée de binaires. Afin de tester cette hypothèse, nous avons appliqué les techniques de génération RI analysées à un ensemble fixe de programmes et nous avons comparé les nombre d'instructions RI résultant. Nous avons pris comme référence de comparaison le nombre d'instructions natives présentes dans le programme compilé.

Nous avons vu que, en général, les techniques basées sur le binaire produisaient un plus grand nombre d'instructions RI que la traduction basée sur le source (KLEE) ; bien sûr, il y a beaucoup de facteurs qui impliquent la taille des artefacts de traduction générés. C'est plus significatif lorsque la cible du processus de traduction est la même RI, ce qui supprime une variable de l'analyse. Par conséquent, la comparaison de S2E et McSema est particulièrement intéressante : les deux outils commencent au niveau binaire et produisent de la RI LLVM, nous avons donc pu comparer leurs résultats avec la RI LLVM générée à partir du source par KLEE. Alors que la RI produite à partir du code source était plutôt succincte, et dans presque tous les cas contenant moins d'instructions que le code machine équivalent, la RI correspondante générée par les binaires augmentaient le nombre d'instructions d'un facteur de supérieur à 6 pour S2E et 4,54 pour McSema. De plus, il était intéressant de voir que la traduction vers VEX IR entraînait une augmentation du nombre d'instructions qui était similaire aux outils binaires traduisant en RI LLVM ; une analyse manuelle suggère que le contenu sémantique des instructions dans la RI VEX est comparable à la RI LLVM. Une expérience sur l'architecture ARM a globalement confirmé ces résultats sur une architecture différente. En moyenne, nous avons constaté que la RI générée à partir de binaires était considérablement plus grand que la RI basée sur le source.

Vitesse d'exécution

Un autre aspect de l'exécution symbolique qui nous intéressait était de savoir dans quelle mesure la RI générée est adaptée à l'exécution. Il existe un large spectre de choix entre QSYM, qui renonce à la traduction en RI et exécute directement le code machine instrumenté, calculant des contraintes symboliques à la volée, et KLEE, qui interprète une RI de haut niveau dérivée du code source. Intuitivement, nous nous attendrions à ce qu'une RI proche (ou identique à) du code machine soit efficace à exécuter, tandis qu'une représentation plus abstraite soit plus adaptée à l'analyse statique mais plus lente en exécution.

Afin d'évaluer la vitesse d'exécution, nous avons compté le nombre d'instructions exécutés au niveau RI et le temps consacré à cette exécution lors de la conduite nos expériences. Le résultat était une quantité que nous appelions *taux d'exécution* ; il représente le nombre d'instructions exécutées par unité de temps. Nous avons observé que QSYM exécute son « RI » le plus rapidement, suivi de KLEE, S2E et angr. Cela correspondait à notre intuition, étant donné que QSYM réalise l'exécution symbolique directement sur l'assembleur. KLEE et S2E partagent une base commune, mais tandis que KLEE exécute une RI très concise, S2E a beaucoup plus d'instructions à interpréter. De plus, S2E doit générer sa RI à la volée alors que, dans le cas de KLEE, la génération RI est une étape de prétraitement.

Complexité des requêtes

Parallèlement à l'exécution RI, la résolution SMT est l'une des principales charges de travail de l'exécution symbolique. Il est donc important d'explorer l'impact de la génération de la RI sur la complexité des requêtes SMT lors de l'exécution. Intuitivement, si la RI porte beaucoup d'informations sémantiques, il devrait être possible pour le moteur d'exécution symbolique de formuler des requêtes succinctes.

Nous avons mesuré le temps mis par Z3 (un solveur SMT populaire) pour résoudre toutes les requêtes enregistrées dans nos expériences. Nous appelons le nombre de requêtes traitées par fois le taux de requêtes. Nous avons observé que angr et QSYM affichaient des taux de requêtes plus faibles à KLEE, dont le taux médian était significativement plus élevé. Les requêtes de S2E entraient dans une plage similaire à celle de KLEE (qui est raisonnable car S2E est basé sur KLEE), mais la médiane de S2E était considérablement plus faible et similaire à angr et Qsym. En général, il semblait que les trois systèmes d'exécution symboliques binaires généraient des requêtes plus difficiles que KLEE, le système basé sur le source.

B.2.5 *Conclusion*

En résumé, nous avons constaté ce qui suit :

1. Pour la taille du code, le facteur le plus important est de savoir si la RI est générée à partir de code source ou de binaires. Alors que la RI basée sur la source est souvent plus succincte que le code machine, la RI binaire a tendance à gonfler le code d'un facteur compris entre 3 et 7.

2. Nous n'avons pas observé de différences significatives de vitesse d'exécution entre la RI LLVM et la RI VEX. QSYM, cependant, gagne un avantage distinct dans la vitesse en se dispensant d'une RI traditionnelle et instrumentant le code machine directement, au détriment de portabilité.

3. Lorsqu'elles sont générées à partir du code machine, la RI LLVM et la RI VEX conduisent à des requêtes de complexité similaire ; les requêtes dérivées directement du code machine sont relativement similaires. S2E génère des requêtes plus simples que angr et QSYM dans certains cas mais le taux de requêtes médian est similaire. Cependant, la RI basé sur le source apparaît pour conduire de manière fiable à des requêtes plus simples lors de l'exécution symbolique.

Étant donné que la vitesse d'exécution la plus élevée est obtenue avec des instructions de *bas niveau*, alors que les meilleures performances du solveur sont obtenues avec des requêtes générées à partir de code de *haut niveau*, nous avons conçu un nouveau moteur d'exécution symbolique avec exactement ces deux propriétés, qui sera décrit dans la section suivante.

B.3 SYMCC : INTÉGRER L'EXÉCUTION SYMBOLIQUE PAR LE COMPILATEUR

Cette section est basée sur le papier « Symbolic execution with SymCC : Don't execute, compile ! », publié dans le 29ième USENIX Security Symposium (USENIX Security 2020) à Boston, MA, USA.

Inspirés par notre comparaison avec les systèmes existants, nous avons proposé une méthode d'exécution alternative et avons montré que cela conduit à une exécution symbolique considérablement plus rapide et une meilleure couverture du programme et donc plus de bugs découverts.

Examinons d'abord la manière dont les moteurs d'exécution symbolique les plus performants sont mise en œuvre. À quelques exceptions notables (qui seront examinées en détail plus loin), la plupart les implémentations traduisent le programme testé en une représentation intermédiaire (par exemple, le bitcode LLVM), qui est ensuite

exécutée symboliquement. Conceptuellement, le système parcourt les instructions du programme cible un par un, effectue les calculs demandés et assure également le suivi des sémantique en termes de toute entrée symbolique. Il s'agit essentiellement d'un interpréteur ! L'interprétation est, en général, moins efficace que la compilation car elle effectue un travail à chaque exécution qu'un compilateur ne doit faire qu'une seule fois [37, 83]. Notre idée centrale est donc d'appliquer la « compilation au lieu de l'interprétation » à l'exécution symbolique pour obtenir de meilleures performances. Mais qu'est-ce que la compilation dans le cadre de l'exécution symbolique ? Dans les langages de programmation, c'est le processus de remplacement des instructions de la langue source par séquences de code machine qui effectuent des actions équivalentes. Donc, pour postuler la même idée à l'exécution symbolique, nous avons *intégré* le traitement symbolique dans le programme cible. Le résultat final est un binaire qui s'exécute sans le besoin d'un interpréteur externe ; il effectue les mêmes actions que le programme cible mais garde en plus les expressions symboliques. Cette technique lui permet d'effectuer tout raisonnement symbolique qui est conventionnellement appliqué par l'interpréteur, tout en conservant la vitesse d'un programme compilé. Nous avons présenté une implémentation de notre idée, appelée SymCC, qui utilise l'infrastructure compilateur LLVM.

B.3.1 *Implémentation*

Dans le contexte de l'exécution symbolique, les approches actuelles interprètent (dans le cas des implémentations basées sur une RI) ou exécutent directement sur le CPU mais avec un observateur attaché (dans les implémentations sans RI), effectuant des calculs qui ne font pas partie du programme cible. De manière informelle, Les approches RI sont faciles à mettre en œuvre et à maintenir mais plutôt lentes, tandis que les techniques sans RI atteignent des performances élevées mais sont complexes à mettre en œuvre. Un de nos objectifs principal est de combiner les avantages des deux approches, c'est-à-dire, construire un système qui est facile à mettre en œuvre mais rapide. Pour ce faire, nous avons compilé la logique de l'interpréteur symbolique (ou observateur) dans le programme cible. Nous décrivons maintenant certains aspects fondamentaux de notre conception.

Gestionnaires symboliques

Le cœur de notre transformation au moment de la compilation est l'insertion d'appels à gérer des calculs symboliques. Nous avons écrit une passe de compilateur personnalisée qui parcourt le bitcode LLVM produit par le frontend du compilateur et insère le code pour la gestion symbolique. Le code inséré appelle les fonctions exportées par le backend symbolique : nous fournissons un wrapper fin autour

du solveur SMT Z3 [23], ainsi que l'intégration optionnelle avec le backend plus sophistiqué de QSYM [86].

Bibliothèque de support

Comme nous compilons des capacités d'exécution symboliques dans le programme cible, tous les composants d'un moteur d'exécution symbolique typique doivent être disponibles. On a donc regroupé le backend symbolique dans une bibliothèque qui est utilisée par le programme cible. La bibliothèque expose les points d'entrée dans le backend symbolique à appeler à partir de la cible instrumentée, par exemple, des fonctions pour construire des expressions symboliques et pour informer le backend des sauts conditionnels.

Contrôles de variables concrètes

Il est important de réaliser que chaque appel a la bibliothèque de support inséré introduit des frais généraux : elle invoque finalement le backend symbolique et peut mettre la charge sur le solveur SMT. Cependant, impliquer le backend symbolique n'est nécessaire lorsqu'un calcul reçoit des entrées symboliques. Il n'est pas nécessaire d'informer le backend de calculs entièrement concrets – ce qui permet d'éviter des calculs inutiles. Par conséquent, dans le code que nous générons, nous omettons les appels au backend symbolique si les données sont connues pour être constantes au moment de la compilation. De plus, dans les autres cas, nous insérons des contrôles d'exécution pour limiter les appels backend aux situations où au moins une entrée d'un calcul est symbolique (et donc le résultat peut l'être aussi). Dans notre implémentation, les expressions symboliques sont représentées comme des pointeurs au moment de l'exécution, et les expressions pour les valeurs concrètes sont des pointeurs nuls. Par conséquent, vérifier le caractère concret d'une expression donnée lors de l'exécution est une simple vérification de pointeur nul.

Memoire de recopie

En général, nous stockons les expressions symboliques associées aux données dans une région recopie en mémoire. Notre bibliothèque de support assure le suivi des allocations de mémoire dans le programme cible et les mappe aux régions recopies contenant les expressions symboliques correspondantes qui sont allouées page par page. Il y a cependant un cas particulier : les expressions correspondant a des variables locales sont stockées dans des variables locales elles-mêmes. Cela signifie qu'ils reçoivent le même traitement que les données régulières lors de la génération du code ; en particulier, l'allocateur de registre du compilateur peut décider de les placer dans les registres de la machine pour accès rapide.

Taille de l'implémentation

La passe du compilateur se compose d'environ 1000 lignes de code C++ ; la bibliothèque de support d'exécution, également écrite en C++, comprend encore 1000 lignes (à l'exclusion de Z3 et du code QSYM facultatif). Cette base de code relativement petite montre que l'approche est conceptuellement simple, diminuant ainsi la probabilité de bugs d'implémentation.

B.3.2 *Évaluation*

Nous avons d'abord analysé les performances de notre système sur des benchmarks synthétiques, permettant des expériences contrôlées avec précision. Ensuite, nous avons évalué notre prototype sur des logiciels du monde réel, démontrant que les avantages que nous avons trouvés dans les benchmarks se traduisent par des avantages à trouver des bugs dans le monde réel.

Pour notre évaluation de performance, nous avons comparé SymCC aux moteurs d'exécution symboliques existants, effectuant trois expériences différentes :

1. Nous avons comparé le temps d'exécution pur, c'est-à-dire l'exécution des programmes cibles à l'intérieur des outils d'exécution symboliques mais sans aucune donnée symbolique.

2. Nous avons analysé le temps d'exécution avec des entrées symboliques.

3. Nous avons comparé la couverture des cas de test générés lors de l'exécution symbolique.

Les cibles de notre comparaison sont KLEE et QSYM. Nous avons opté pour KLEE car, comme SymCC, il fonctionne sur le bitcode LLVM généré à partir du code source ; une différence importante, cependant, est que KLEE *interprète* le bitcode tandis que SymCC *compile* le bitcode avec le code pour le traitement symbolique. La comparaison avec KLEE permet donc d'évaluer la valeur de la compilation dans le contexte de l'exécution symbolique. Le choix de QSYM est largement motivé par sa composante d'exécution rapide. Ses auteurs ont démontré des avantages considérables par rapport à d'autres implémentations, et nos propres travaux fournissent des preuves supplémentaires de la notion selon laquelle le composant d'exécution de QSYM atteint des performances élevées par rapport à plusieurs systèmes à l'état de l'art. De plus, notre réutilisation du backend symbolique de QSYM dans SymCC permet une comparaison équitable des composants d'exécution des deux systèmes (c'est-à-dire, leurs frontends).

En comparant le temps d'exécution pur, nous avons constaté que SymCC exécute la plupart des programmes en moins d'une seconde

(et donc presque aussi rapidement que les programmes natifs non instrumentés), tandis que QSYM et KLEE ont besoin de quelques secondes à quelques minutes. Lors de l'exécution avec une entrée symbolique (c'est-à-dire, une tentative de générer, à chaque branchement, des entrées qui entraîneraient l'exécution sur le chemin alternatif), SymCC est considérablement plus rapide que QSYM et KLEE. Enfin, nous avons comparé la couverture du code et constaté que SymCC atteint généralement un niveau de couverture plus élevé que KLEE ; nous attribuons principalement des différences aux backends symboliques significativement différents. De plus, la couverture de SymCC est comparable à celle de QSYM, c'est-à-dire que le composant d'exécution basé sur la compilation fournit des informations de qualité comparable au backend symbolique. Nous avons conclu que l'exécution symbolique basée sur la compilation est beaucoup plus rapide que l'exécution symbolique basée sur une RI et même sans RI dans nos benchmarks tout en obtenant une couverture de code similaire et en maintenant une implémentation simple.

Ensuite, nous avons démontré que ces résultats s'appliquent également à l'analyse de logiciels réels. En particulier, nous avons montré que SymCC atteint des performances globales comparables ou meilleures malgré sa mise en œuvre simple et son approche indépendante de l'architecture du processeur. Nous avons utilisé QSYM et SymCC en combinaison avec l'outil de fuzzing AFL [88] pour tester des projets open source populaires ; il n'a pas été possible de réaliser les mêmes comparaisons avec KLEE en raison d'instructions non prises en charge dans les programmes cibles. Lors de l'expérience, nous avons mesuré la couverture de code vue par AFL et le temps passé sur chaque exécution symbolique du programme cible. Nous avons constaté que SymCC s'exécute non seulement plus rapidement que QSYM, mais atteint également une couverture considérablement plus élevée sur les trois programmes de test. Fait intéressant, le gain de couverture semble être corrélé à l'amélioration de la vitesse, ce qui confirme notre intuition selon laquelle l'accélération de l'exécution symbolique conduit à de meilleurs tests de programme.

Au cours de nos expériences avec la bibliothèque OpenJPEG, SymCC a trouvé deux vulnérabilités qui affectaient la dernière version principale au moment de la rédaction ainsi que les versions précédentes publiées. Les deux vulnérabilités écrivaient des débordements de tampon de tas et étaient donc probablement exploitables. Ces vulnérabilités n'avaient pas été détectés précédemment, malgré que OpenJPEG soit testé régulièrement par des fuzzers à l'état de l'art et avec des ressources de calcul considérables par le projet OSS-Fuzz de Google. Nous avons signalé les vulnérabilités aux responsables du projet, qui ont confirmé et corrigé les deux. Les vulnérabilités se sont ensuite vu attribuer les identifiants CVE 2020-6851 et 2020-8112 et ont reçu des scores d'impact élevés par le NIST (7,5 et 8,8, respectivement). Dans

les deux cas, les problèmes provenaient de vérifications de limites manquantes ou incorrectes – l'exécution symbolique a été en mesure d'identifier le problème potentiel et de résoudre les contraintes correspondantes afin de générer des entrées qui déclenchent le problème. Dans les mêmes expériences, QSYM n'a pas trouvé de nouvelles vulnérabilités.

B.3.3 *Conclusion*

Nous avons présenté SymCC, un système d'exécution symbolique qui intègre des capacités de traitement symbolique dans les programmes testés via un compilateur. L'évaluation a montré que l'incorporation directe entraîne des améliorations significatives de la vitesse d'exécution des programmes cibles, surpassant largement les approches actuelles. Une exécution plus rapide accélère l'analyse dans son ensemble et augmente les chances de découverte de bugs, nous amenant à trouver deux vulnérabilités à fort impact dans une bibliothèque largement testée. En utilisant un compilateur pour insérer la gestion symbolique dans les programmes cibles, nous avons combiné les avantages de l'exécution symbolique basée sur une RI et sans RI : SymCC est indépendant de l'architecture et peut prendre en charge divers langages de programmation avec peu d'effort d'implémentation (comme les approches basées sur RI), mais l'analyse est très rapide – considérablement plus rapide même que les techniques actuelles sans RI.

B.4 SYMQEMU : INTÉGRER L'EXÉCUTION SYMBOLIQUE PAR UN ÉMULATEUR

Les travaux décrits dans cette section seront soumis pour publication au Network and Distributed System Security Symposium (NDSS).

Une caractéristique importante des systèmes d'exécution symbolique est de savoir s'ils requièrent le code source du programme testé (comme SymCC) ou s'appliquent à la place aux programmes binaires uniquement de manière boîte noire. Alors que les tests basés sur la source sont suffisants lorsque l'on teste ses propres produits ou logiciels open source, de nombreux scénarios du monde réel nécessitent la capacité d'analyser les binaires sans le code source disponible :

- Nous sommes de plus en plus entourés et comptons sur des appareils embarqués, dont le micrologiciel est généralement disponible uniquement sous forme binaire. Les audits de sécurité nécessitent donc des outils d'analyse binaires.
- Même lors du test de ses propres produits, les dépendances de bibliothèques propriétaires peuvent ne pas être fournies avec le code source, ce qui rend les approches basées sur la source irréalisables.

- Les tests basés sur la source peuvent tout simplement ne pas être pratiques pour les grands programmes testés. Avec un outil basé sur la source, il est généralement nécessaire de créer toutes les dépendances de bibliothèque d'une manière dédiée prescrite par l'outil, ce qui peut imposer une lourde charge au testeur. De plus, si le programme testé est implémenté dans un mélange de langages de programmation, il est probable que les outils basés sur la source ne puissent pas tous les gérer.

Lorsqu'un exécuteur symbolique uniquement binaire est requis, les utilisateurs sont souvent confrontés à un dilemme : les outils optimisent soit pour les performances soit pour l'indépendance de l'architecture mais fournissent rarement les deux. Par exemple, QSYM [86] a récemment montré comment implémenter une exécution symbolique très rapide des binaires, mais il atteint sa vitesse élevée en liant l'implémentation au jeu d'instructions des processeurs x86. Non seulement cela rend le système dépendant de l'architecture, mais cela augmente également sa complexité en raison de la taille des jeux d'instructions des processeurs modernes ; selon les propres mots des auteurs, leur approche consiste à « payer pour la complexité de la mise en œuvre afin de réduire les frais d'exécution ». En revanche, S2E [17] est un exemple d'un système qui est largement applicable mais souffre de performances relativement inférieures. S2E peut analyser conceptuellement le code de la plupart des architectures CPU, y compris le code du noyau. Cependant, sa large applicabilité est achetée avec plusieurs traductions du programme cible, ce qui augmente la complexité du système et affecte finalement les performances. En fait, il apparaît que des performances élevées en analyse symbolique uniquement binaire sont souvent obtenues avec des implémentations hautement spécialisées – un choix de conception qui est en conflit avec la flexibilité architecturale.

Nous avons montré une alternative qui (a) est indépendante de l'architecture cible du programme testé, (b) a une faible complexité d'implémentation, mais (c) atteint des performances élevées. L'idée clé de notre système, SymQEMU, est que l'émulation de plate-forme de QEMU [5] peut être combinée avec un mécanisme très léger pour l'exécution symbolique : au lieu d'une traduction coûteuse en calcul du programme cible vers une représentation intermédiaire qui est ensuite interprétée symboliquement (comme dans S2E), nous nous connectons au mécanisme de traduction binaire de QEMU afin de compiler la gestion symbolique directement dans le code machine que l'émulateur émet et exécute (comme SymCC, mais pour les binaires). Cette approche donne des performances similaires à QSYM tout en conservant une indépendance totale de la plate-forme.

B.4.1 *Implémentation*

Nous présentons maintenant la conception et la mise en œuvre de notre exécuteur symbolique binaire SymQEMU. Il s'appuie sur les travaux antérieurs et combine les avantages des systèmes de pointe avec de nouvelles idées pour créer un moteur d'analyse rapide mais flexible. Le système a deux objectifs principaux :

1. Obtenir des performances élevées afin de s'adapter aux logiciels du monde réel.

2. Rester raisonnablement indépendant de la plateforme, c'est-à-dire que l'ajout de la prise en charge d'une architecture de processeur ne devrait pas nécessiter d'effort majeur.

Nous avons vu que les solutions actuelles atteignent l'indépendance de la plate-forme en traduisant le programme testé en une représentation intermédiaire – de cette façon, afin de prendre en charge une nouvelle architecture, seul le traducteur doit être porté. Idéalement, on choisit un langage intermédiaire pour lequel des traducteurs de nombreuses architectures pertinentes existent déjà. Représenter des programmes d'une manière indépendante de l'architecture pour plus de flexibilité est une technique bien connue qui a été appliquée avec succès dans de nombreux autres domaines, par exemple, la conception de compilateurs [49] ou l'analyse binaire statique [42]. Nous l'avons donc également intégré à notre conception.

Bien que la traduction de programmes vers une représentation intermédiaire nous donne de la flexibilité, nous devons être conscients de l'impact sur les performances : la traduction statique de programmes binaires est difficile car le désassemblage peut ne pas être fiable et l'exécution de la traduction au moment de l'exécution entraîne des frais généraux lors de l'analyse. Nous pensons que c'est la raison principale pour laquelle les exécuteurs symboliques traducteurs comme S2E et angr ont des performances bien en dessous des systèmes qui ne sont pas basés sur une traduction, comme QSYM. Alors, comment pouvons-nous construire un système de traduction qui fonctionne toujours bien ? Notre approche est la suivante :

1. Traduire le programme cible dans une langue intermédiaire au moment de l'exécution.

2. Instrumenter la représentation intermédiaire comme nécessaire pour l'exécution symbolique.

3. Compiler la représentation intermédiaire en code machine adapté au CPU exécutant l'analyse et l'exécuter directement.

En compilant le programme cible instrumenté en code machine, nous compensons la pénalité de performance encourue par la traduction du binaire dans un langage intermédiaire en premier lieu : le CPU

exécute le code machine beaucoup plus rapidement qu'un interpréteur ne peut exécuter la représentation intermédiaire, de sorte que nous atteignons performances comparables à un système sans traduction tout en conservant l'avantage de l'indépendance de l'architecture qui accompagne la traduction de programme.

Concrètement, nous avons étendu un composant dans QEMU appelé TCG (pour « Tiny Code Generator »). Dans QEMU non modifié, TCG est responsable de la traduction des blocs de code machine d'architecture invitée en un langage indépendant de l'architecture appelé TCG ops, puis compile ces opérations TCG en code machine pour l'architecture hôte. Les blocs traduits sont ensuite mis en cache pour des raisons de performances, la traduction ne doit donc se produire qu'une seule fois. SymQEMU insère une étape de plus dans le processus : pendant que le programme testé est en cours de conversion en opérations TCG, nous émettons non seulement les instructions qui émulent le processeur invité, mais ajoutons des opérations TCG supplémentaires pour construire des expressions symboliques pour les résultats. Par exemple, supposons que le programme cible ajoute deux octets d'entrée. QEMU traduirait l'instruction de code machine à ajouter en opérations TCG qui chargent les opérandes à partir des registres ou de la mémoire du CPU émulé, effectuent l'addition et stockent le résultat dans le registre cible ; il traduirait ensuite ces opérations TCG en code machine et les exécuterait sur le CPU hôte. SymQEMU effectue la même traduction, mais insère en outre des opérations TCG qui appellent une bibliothèque de support (la même que dans SymCC) afin de construire une expression symbolique représentant l'addition des deux octets d'entrée. Fondamentalement, SymQEMU n'effectue pas ces appels à la bibliothèque de support (comme le ferait un interpréteur) – il n'émet que les opérations TCG correspondantes et s'appuie sur les mécanismes QEMU normaux pour les traduire en code machine. De cette façon, les formules symboliques sont construites en code machine natif sans encourir la surcharge associée à l'interprétation d'un langage intermédiaire. Globalement, SymQEMU ajoute environ 2000 lignes de code C à QEMU.

B.4.2 *Évaluation*

Afin d'évaluer SymQEMU, nous avons effectué deux ensembles d'expériences différents :

1. Nous l'avons comparé à un certain nombre de fuzzers à l'état de l'art à l'aide de Google FuzzBench.

2. Étant donné que FuzzBench n'inclut pas d'outils d'exécution symbolique, nous avons effectué une comparaison avec les exécuteurs symboliques populaires uniquement binaires sur un ensemble de programmes du monde réel.

Google a annoncé FuzzBench en mars 2020 en tant que service gratuit entièrement automatisé et open source pour évaluer les fuzzers. Il teste les fuzzers dans un environnement contrôlé, en comparant leurs performances sur un grand nombre de cibles issues de Google OSS-Fuzz, une collection de cibles fuzz pour les logiciels open source. Pour chaque cible, le service compare la couverture obtenue par les fuzzers. L'intégration d'un nouvel outil d'analyse revient à configurer un conteneur Docker pour configurer l'environnement, créer les programmes cibles et lancer l'analyse. Nous avons ajouté une combinaison de SymQEMU et AFL à l'ensemble d'outils d'analyse, et l'équipe FuzzBench a gracieusement effectué une série d'expériences. Au total, ils ont exécuté SymQEMU et 16 configurations de fuzzer sur 21 cibles pendant 24 heures, effectuant 15 essais par fuzzer et cible. En moyenne dans toutes les expériences, SymQEMU a surperformé tous les fuzzers sauf Honggfuzz, 6 d'entre eux avec une signification statistique. Sur 3 des 21 cibles, SymQEMU a atteint la couverture la plus élevée de tous les outils. Nous considérons cela comme une confirmation de la puissance de SymQEMU.

Nous avons également comparé SymQEMU à des outils de pointe dans le domaine de l'exécution symbolique. En particulier, nous l'avons évalué par rapport à QSYM, un exécuteur symbolique très rapide qui peut fonctionner avec des binaires x86 arbitraires. Nous avons omis angr de la comparaison car les expériences préliminaires ont montré que sa vitesse d'exécution est nettement inférieure à celle des autres outils ; angr privilégie la polyvalence et la facilité d'utilisation interactive par rapport à la vitesse brute. Nous avons constaté que SymQEMU surpasse QSYM par une grande marge en termes de couverture de code et de temps d'exécution.

B.5 CONCLUSION

Dans ma thèse, j'examine d'abord l'état de l'art dans l'exécution symbolique avec un accent particulier sur la performance. Sur la base de cette étude, je développe une approche basée sur un compilateur pour l'exécution symbolique qui fonctionne bien mieux que les systèmes actuels. Enfin, j'applique une technique similaire à l'exécution symbolique uniquement binaire, montrant que l'incorporation directe de la gestion symbolique peut conduire à des gains de performances élevés. J'espère que mon travail a contribué à faire de l'exécution symbolique un outil pratique dans le développement, le test et l'analyse de logiciels.

BIBLIOGRAPHY

[1] Apple Inc. *Swift.org – Compiler and Standard Library*. https://swift.org/compiler-stdlib/#compiler-architecture.

[2] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. "REDQUEEN: Fuzzing with Input-to-State Correspondence." In: *Network and Distributed System Security Symposium (NDSS)*. Vol. 19. 2019, pp. 1–15.

[3] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. "A survey of symbolic execution techniques." In: *ACM Computing Surveys (CSUR)* 51.3 (2018), p. 50.

[4] Clark Barrett, Aaron Stump, and Cesare Tinelli. "The SMT-LIB standard: Version 2.0." In: *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, England)*. Vol. 13. 2010, p. 14.

[5] Fabrice Bellard. "QEMU, a fast and portable dynamic translator." In: *USENIX Annual Technical Conference, FREENIX Track*. Vol. 41. 2005, p. 46.

[6] Rastislav Bodík, Kartik Chandra, Phitchaya Mangpo Phothilimthana, and Nathaniel Yazdani. "Domain-specific symbolic compilation." In: *2nd Summit on Advances in Programming Languages (SNAPL 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. 2017.

[7] 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*. ACM. 2017, pp. 2329–2344.

[8] Ella Bounimova, Patrice Godefroid, and David Molnar. "Billions and Billions of Constraints: Whitebox Fuzz Testing in Production." In: *Proceedings of the 2013 International Conference on Software Engineering*. ICSE '13. San Francisco, CA, USA: IEEE Press, 2013, pp. 122–131. ISBN: 978-1-4673-3076-3. URL: http://dl.acm.org/citation.cfm?id=2486788.2486805.

[9] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J Schwartz. "BAP: A binary analysis platform." In: *International Conference on Computer Aided Verification*. Springer. 2011, pp. 463–469.

[10] Robert Brummayer and Armin Biere. "Boolector: An efficient SMT solver for bit-vectors and arrays." In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2009, pp. 174–177.

[11] Stefan Bucur, Vlad Ureche, Cristian Zamfir, and George Candea. "Parallel symbolic execution for automated real-world software testing." In: *Proceedings of the 6th ACM SIGOPS/EuroSys Conference on Computer Systems*. ACM. 2011, pp. 183–198.

[12] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs." In: *OSDI*. Vol. 8. 2008, pp. 209–224.

[13] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. "EXE: automatically generating inputs of death." In: *ACM Transactions on Information and System Security (TISSEC)* 12.2 (2008), p. 10.

[14] Giovanni Camurati, Sebastian Poeplau, Marius Muench, Tom Hayes, and Aurélien Francillon. "Screaming Channels: When Electromagnetic Side Channels Meet Radio Transceivers." In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM. 2018, pp. 163–177.

[15] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. "Unleashing Mayhem on binary code." In: *2012 IEEE Symposium on Security and Privacy*. IEEE. 2012, pp. 380–394.

[16] 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*. ACM. 2018, pp. 2095–2108.

[17] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. "S2E: A platform for in-vivo multi-path analysis of software systems." In: *ACM SIGARCH Computer Architecture News*. Vol. 39. 1. ACM. 2011, pp. 265–278.

[18] Jaeseung Choi, Joonun Jang, Choongwoo Han, and Sang Kil Cha. "Grey-box concolic testing on binary code." In: *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE. 2019, pp. 736–747.

[19] Cristina Cifuentes and K. John Gough. "Decompilation of binary programs." In: *Software: Practice and Experience* 25.7 (1995), pp. 811–829.

[20] Peter Collingbourne, Cristian Cadar, Paul H.J. Kelly, et al. "Symbolic crosschecking of floating-point and SIMD code." In: *European Conference on Computer Systems (EuroSys 2011)*. 2011.

[21] Nassim Corteggiani, Giovanni Camurati, and Aurélien Francillon. "Inception: system-wide security testing of real-world embedded systems software." In: *27th USENIX Security Symposium (USENIX Security 18)*. 2018, pp. 309–326.

[22] Nassim Corteggiani, Giovanni Camurati, Marius Muench, Sebastian Poeplau, and Aurélien Francillon. "SoC Security Evaluation: Reflections on Methodology and Tooling." In: *IEEE Design & Test* (2020).

[23] Leonardo De Moura and Nikolaj Bjørner. "Z3: An efficient SMT solver." In: *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2008, pp. 337–340.

[24] Artem Dinaburg and Andrew Ruef. "McSema: Static translation of x86 instructions to LLVM." In: *ReCon 2014 Conference, Montreal, Canada*. 2014.

[25] Joe W. Duran and Simeon Ntafos. "A Report on Random Testing." In: *Proceedings of the 5th International Conference on Software Engineering*. ICSE '81. San Diego, California, USA: IEEE Press, 1981, pp. 179–183. ISBN: 0-89791-146-6. URL: http://dl.acm.org/citation.cfm?id=800078.802530.

[26] E. Allen Emerson and Edmund M. Clarke. "Characterizing correctness properties of parallel programs using fixpoints." In: *International Colloquium on Automata, Languages, and Programming*. Springer. 1980, pp. 169–181.

[27] Andrea Fioraldi, Daniele Cono D'Elia, and Leonardo Querzoni. "Fuzzing Binaries for Memory Safety Errors with QASan." In: (). URL: https://andreafioraldi.github.io/assets/qasan-secdev20.pdf.

[28] Free Software Foundation. *Coreutils - GNU core utilities*. https://www.gnu.org/software/coreutils/. 2016.

[29] 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*. IEEE. 2018, pp. 679–696.

[30] Vijay Ganesh and David L Dill. "A decision procedure for bit-vectors and arrays." In: *International Conference on Computer Aided Verification*. Springer. 2007, pp. 519–531.

[31] Go git repositories. *gollvm*. https://go.googlesource.com/gollvm/.

[32] Patrice Godefroid, Nils Klarlund, and Koushik Sen. "DART: directed automated random testing." In: *ACM Sigplan Notices*. Vol. 40. 6. ACM. 2005, pp. 213–223.

[33] Patrice Godefroid, Michael Y. Levin, and David Molnar. "SAGE: whitebox fuzzing for security testing." In: *Communications of the ACM* 55.3 (2012), pp. 40–44.

[34] Eric Gustafson et al. "Toward the Analysis of Embedded Firmware through Automated Re-hosting." In: *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)*. Chaoyang District, Beijing: USENIX Association, Sept. 2019, pp. 135–150. ISBN: 978-1-939133-07-6. URL: https://www.usenix.org/conference/raid2019/presentation/gustafson.

[35] Alex Horn. *Clang CRV Front-end*. https://github.com/ahorn/native-symbolic-execution-clang. 2014.

[36] Alex Horn. *SMT Kit*. https://github.com/ahorn/smt-kit. 2014.

[37] C.-A. Hsieh, M. T. Conte, T. L. Johnson, J. C. Gyllenhaal, and W.-W. Hwu. "Compilers for improved Java performance." In: *Computer* 30.6 (June 1997), pp. 67–75. DOI: 10.1109/2.587551.

[38] 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)*. Los Alamitos, CA, USA: IEEE Computer Society, May 2020, pp. 1613–1627. DOI: 10.1109/SP40000.2020.00063. URL: https://doi.ieeecomputersociety.org/10.1109/SP40000.2020.00063.

[39] Timotej Kapus and Cristian Cadar. "Automatic Testing of Symbolic Execution Engines via Program Generation and Differential Testing." In: *IEEE/ACM International Conference on Automated Software Engineering (ASE 2017)*. Urbana-Champaign, IL, USA, Nov. 2017, pp. 590–600.

[40] Timotej Kapus, Oren Ish-Shalom, Shachar Itzhaky, Noam Rinetzky, and Cristian Cadar. "Computing Summaries of String Loops in C for Better Testing and Refactoring." In: *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2019)*. Phoenix, AZ, USA, June 2019, pp. 874–888.

[41] Timotej Kapus, Martin Nowack, and Cristian Cadar. "Constraints in Dynamic Symbolic Execution: Bitvectors or Integers?" In: *International Conference on Tests and Proofs*. Springer. 2019, pp. 41–54.

[42] Soomin Kim, Markus Faerevaag, Minkyu Jung, SeungIl Jung, DongYeop Oh, JongHyup Lee, and Sang Kil Cha. "Testing intermediate representations for binary analysis." In: *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. IEEE Press. 2017, pp. 353–364.

[43] James C. King. "Symbolic execution and program testing." In: *Communications of the ACM* 19.7 (1976), pp. 385–394.

[44] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. "Evaluating fuzz testing." In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 2018, pp. 2123–2138.

[45] Gergely Kovásznai, Helmut Veith, Andreas Fröhlich, and Armin Biere. "On the complexity of symbolic verification and decision problems in bit-vector logic." In: *International Symposium on Mathematical Foundations of Computer Science*. Springer. 2014, pp. 481–492.

[46] Volodymyr Kuznetsov, Johannes Kinder, Stefan Bucur, and George Candea. "Efficient state merging in symbolic execution." In: *Acm Sigplan Notices*. Vol. 47. 6. ACM. 2012, pp. 193–204.

[47] LLVM Project. *libFuzzer – a library for coverage-guided fuzz testing*. https://llvm.org/docs/LibFuzzer.html.

[48] LLVM Project. *"libc++" C++ Standard Library*. https://libcxx.llvm.org/.

[49] Chris Lattner and Vikram Adve. "LLVM: A compilation framework for lifelong program analysis & transformation." In: *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*. IEEE Computer Society. 2004, p. 75.

[50] Lixin Li and Chao Wang. "Dynamic analysis and debugging of binary code for security applications." In: *International Conference on Runtime Verification*. Springer. 2013, pp. 403–423.

[51] Tianhai Liu, Mateus Araújo, Marcelo d'Amorim, and Mana Taghdiri. "A comparative study of incremental constraint solving approaches in symbolic execution." In: *Haifa Verification Conference*. Springer. 2014, pp. 284–299.

[52] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. "Pin: building customized program analysis tools with dynamic instrumentation." In: *Acm sigplan notices*. Vol. 40. 6. ACM. 2005, pp. 190–200.

[53] Valentin J. M. Manès, Soomin Kim, and Sang Kil Cha. "Ankou: Guiding Grey-box Fuzzing towards Combinatorial Difference." In: *Proceedings of the International Conference on Software Engineering*. 2020, pp. 1024–1036.

[54] Paul Dan Marinescu and Cristian Cadar. "KATCH: High-coverage testing of software patches." In: *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. 2013, pp. 235–245.

[55] Marius Muench, Jan Stijohann, Frank Kargl, Aurélien Francillon, and Davide Balzarotti. "What you corrupt is not what you crash: Challenges in fuzzing embedded devices." In: *NDSS 2018, Network and Distributed Systems Security Symposium, 18-21 February 2018, San Diego, CA, USA*. 2018.

[56] Nicholas Nethercote and Julian Seward. "Valgrind: a framework for heavyweight dynamic binary instrumentation." In: *ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI 2007)*. Vol. 42. 6. ACM. 2007, pp. 89–100.

[57] Anh Nguyen-Tuong, David Melski, Jack W. Davidson, Michele Co, William Hawkins, Jason D. Hiser, Derek Morris, Ducson Nguyen, and Eric Rizzi. "Xandra: An Autonomous Cyber Battle System for the Cyber Grand Challenge." In: *IEEE Security & Privacy* 16.2 (2018), pp. 42–51.

[58] Sebastian Österlund, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. "ParmeSan: Sanitizer-guided Greybox Fuzzing." In: *29th USENIX Security Symposium (USENIX Security 20)*. 2020.

[59] Hristina Palikareva and Cristian Cadar. "Multi-solver support in symbolic execution." In: *International Conference on Computer Aided Verification*. Springer. 2013, pp. 53–68.

[60] Chengbin Pang, Ruotong Yu, Yaohui Chen, Eric Koskinen, Georgios Portokalidis, Bing Mao, and Jun Xu. "SoK: All You Ever Wanted to Know About x86/x64 Binary Disassembly But Were Afraid to Ask." In: *arXiv preprint arXiv:2007.14266* (2020).

[61] Sebastian Poeplau and Aurélien Francillon. "Systematic comparison of symbolic execution systems: Intermediate representation and its generation." In: *Proceedings of the 35th Annual Computer Security Applications Conference*. ACM. 2019, pp. 163–176.

[62] Sebastian Poeplau and Aurélien Francillon. "SymQEMU: Compilation-based symbolic execution for binaries." Under submission. Sept. 2020.

[63] Sebastian Poeplau and Aurélien Francillon. "Symbolic execution with SymCC: Don't interpret, compile!" In: *29th USENIX Security Symposium (USENIX Security 20)*. Distinguished Paper Award. Boston, MA: USENIX Association, Aug. 2020. URL: https://www.usenix.org/conference/usenixsecurity20/presentation/poeplau.

[64] Xiao Qu and Brian Robinson. "A case study of concolic testing tools and their limitations." In: *2011 International Symposium on Empirical Software Engineering and Measurement*. IEEE. 2011, pp. 117–126.

[65] Jean-Pierre Queille and Joseph Sifakis. "Specification and verification of concurrent systems in CESAR." In: *International Symposium on Programming*. Springer. 1982, pp. 337–351.

[66] Nguyen Anh Quynh and Dang Hoang Vu. *Unicorn – The ultimate CPU emulator*. https://www.unicorn-engine.org/. 2015.

[67] David A. Ramos and Dawson Engler. "Under-constrained symbolic execution: Correctness checking for real code." In: *24th USENIX Security Symposium (USENIX Security 15)*. 2015, pp. 49–64.

[68] Eric F Rizzi, Sebastian Elbaum, and Matthew B. Dwyer. "On the techniques we create, the tools we build, and their misalignments: a study of KLEE." In: *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. IEEE. 2016, pp. 132–143.

[69] Florent Saudel and Jonathan Salwan. "Triton: A Dynamic Symbolic Execution Framework." In: *Symposium sur la sécurité des technologies de l'information et des communications, SSTIC, France, Rennes, June 3-5 2015*. SSTIC, 2015, pp. 31–54.

[70] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. "All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask)." In: *2010 IEEE Symposium on Security and Privacy*. IEEE. 2010, pp. 317–331.

[71] Koushik Sen, Darko Marinov, and Gul Agha. "CUTE: a concolic unit testing engine for C." In: *ACM SIGSOFT Software Engineering Notes*. Vol. 30. 5. ACM. 2005, pp. 263–272.

[72] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. "AddressSanitizer: A fast address sanity checker." In: *Presented as part of the 2012 USENIX Annual Technical Conference (USENIX ATC 12)*. 2012, pp. 309–318.

[73] Yan Shoshitaishvili, Ruoyu Wang, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. "Firmalice – Automatic Detection of Authentication Bypass Vulnerabilities in Binary Firmware." In: *Network and Distributed System Security Symposium (NDSS)*. 2015.

[74] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, et al. "SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis." In: *2016 IEEE Symposium on Security and Privacy*. IEEE. 2016, pp. 138–157.

[75] Dokyung Song, Julian Lettner, Prabhu Rajasekaran, Yeoul Na, Stijn Volckaert, Per Larsen, and Michael Franz. "SoK: Sanitizing for Security." In: *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2019, pp. 1275–1295.

[76] Evgeniy Stepanov and Konstantin Serebryany. "MemorySanitizer: fast detector of uninitialized memory use in C++." In: *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization*. IEEE Computer Society. 2015, pp. 46–55.

[77] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. "Driller: Augmenting Fuzzing Through Selective Symbolic Execution." In: *NDSS*. Vol. 16. 2016, pp. 1–16.

[78] The Clang Team. *Clang C Language Family Frontend for LLVM*. https://clang.llvm.org/. 2019.

[79] The Rust Programming Language Team. *Guide to rustc development*. https://rust-lang.github.io/rustc-guide/. 2019.

[80] Emina Torlak and Rastislav Bodik. "A lightweight symbolic virtual machine for solver-aided host languages." In: *ACM SIGPLAN Notices*. Vol. 49. 6. ACM. 2014, pp. 530–541.

[81] Trail of Bits. *Manticore: Symbolic execution for humans*. https://blog.trailofbits.com/2017/04/27/manticore-symbolic-execution-for-humans/. 2017.

[82] Trail of Bits. *Binary symbolic execution with KLEE-Native*. https://blog.trailofbits.com/2019/08/30/binary-symbolic-execution-with-klee-native/. 2019.

[83] Clark Wiedmann. "A Performance Comparison Between an APL Interpreter and Compiler." In: *SIGAPL APL Quote Quad* 13.3 (Mar. 1983), pp. 211–217. ISSN: 0163-6006. DOI: 10.1145/390005.801219. URL: http://doi.acm.org/10.1145/390005.801219.

[84] Hui Xu, Zirui Zhao, Yangfan Zhou, and Michael R. Lyu. "Benchmarking the Capability of Symbolic Execution Tools with Logic Bombs." In: *IEEE Transactions on Dependable and Secure Computing* (2018).

[85] Hui Xu, Yangfan Zhou, Yu Kang, and Michael R. Lyu. "Concolic execution on small-size binaries: challenges and empirical study." In: *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE. 2017, pp. 181–188.

[86] 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)*. 2018, pp. 745–761.

[87] Jonas Zaddach, Luca Bruno, Aurélien Francillon, and Davide Balzarotti. "AVATAR: A Framework to Support Dynamic Security Analysis of Embedded Systems' Firmwares." In: *Network and Distributed System Security Symposium (NDSS)*. Vol. 14. 2014, pp. 1–16.

[88] Michał Zalewski. *american fuzzy lop*. http://lcamtuf.coredump.cx/afl/.

[89] Lintao Zhang, Conor F. Madigan, Matthew H. Moskewicz, and Sharad Malik. "Efficient conflict driven learning in a boolean satisfiability solver." In: *Proceedings of the 2001 IEEE/ACM international conference on Computer-aided design*. IEEE Press. 2001, pp. 279–285.