Systematic Comparison of Symbolic Execution Systems: Intermediate Representation and its Generation

Sebastian Poeplau
Aurélien Francillon
sebastian.poeplau@eurecom.fr
aurelien.francillon@eurecom.fr
EURECOM
Sophia Antipolis, France

ABSTRACT

Symbolic execution has become a popular technique for software testing and vulnerability detection. Most implementations 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 feel that there is little scientific evidence backing the answers to those questions. Therefore, we first develop a methodology for systematic comparison of different approaches to symbolic execution; we then use it to evaluate the impact of the choice of IR and IR generation. We make our comparison framework available to the community for future research.

CCS CONCEPTS

• **Security and privacy** → *Software security engineering*; • **Software and its engineering** → *Software testing and debugging*.

KEYWORDS

symbolic execution, intermediate representation

ACM Reference Format:

Sebastian Poeplau and Aurélien Francillon. 2019. Systematic Comparison of Symbolic Execution Systems: Intermediate Representation and its Generation. In *2019 Annual Computer Security Applications Conference (ACSAC '19), December 9–13, 2019, San Juan, PR, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3359789.3359796

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ACSAC '19, December 9–13, 2019, San Juan, PR, USA

© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 978-1-4503-7628-0/19/12…$15.00
https://doi.org/10.1145/3359789.3359796

1 INTRODUCTION

Symbolic execution has gained popularity as a means of exploring software states without predefined inputs, leading to an increase of the coverage of the tested program, which is, e.g., very attractive for bug finding. Conceptually, a symbolic execution engine keeps track of how each intermediate value is computed while executing a program. Whenever the program hits a conditional statement, the symbolic execution engine can pass the collected information to a solver in order to generate program 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 [1]. 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 [39, 43]. 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, and the only task of the symbolic execution engine is to generate relevant new test inputs as quickly as possible. It is therefore of utmost importance to obtain a better understanding of how the various design decisions in symbolic execution affect its performance.

Typically, symbolic execution engines translate the program under test to an *intermediate representation (IR)* which they can subsequently execute symbolically. Generating the IR from binary 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 [4]. 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 [14]. However, to the best of our knowledge, there is no systematic study backing such claims. The goal of our work is therefore to systematically assess how the choice of IR, and the process of generating it, influence various aspects of symbolic execution.

We select several popular implementations, each with their own mechanism for IR generation, and compare them to discern the

effect on their relative performance. In particular, we aim at answering 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 make our environment and dataset publicly available.

In summary, our contributions 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.
- We publish our setup and data as a common basis for future evaluations.

2 BACKGROUND

This section gives the reader required background information. We focus on the general idea of symbolic execution as well as intermediate representations and SMT solving, before the next section puts the subject of our research in the more general context of designing a symbolic execution engine.

2.1 Symbolic Execution

Symbolic execution was originally proposed by King in 1975 [22]. 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 (e.g., [9]), fueled by significant improvements in *Boolean satisfiability (SAT)* and *satisfiability modulo theories (SMT)* solving [44], and the field continues to be very active to this day.

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 [43], 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. An SMT solver can in many cases solve the expression corresponding to the desired result of the conditional (the so-called *path constraints*) for the input values; in other words, the solver generates inputs that cause the program to run up to the conditional and then take the desired path out of it.

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 [39, 43]. In this latter scenario, a fuzzer selects interesting inputs and symbolic execution merely follows a fixed path dictated by a given concrete program input; 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 [43]. Especially in this hybrid setting, faster symbolic execution amounts to more tested code and—all else being equal—a higher chance of detecting vulnerabilities.

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 files, because the term *lifting* is only appropriate for IR generation that starts from machine code. The choice of IR-generation mechanism has a considerable influence on several aspects of symbolic execution, which is the motivation for this study.

2.3 SMT solving

Symbolic execution engines need to solve path constraints for input values; in other words, they need to solve formulas in the logic of bitvectors and arrays (see Section 5.5 for examples). The field of SMT solving provides tools to address this (generally hard [23]) 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. We conjecture that the way IR is generated has a profound impact on the complexity of the resulting SMT queries.

3 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 recent survey by

Baldoni et al. [1] for a more comprehensive discussion of symbolic execution techniques in general.

Figure 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 [39, 43].

3.1 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 [22]. 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 [20] and the symbolic components of Driller [39] and Qsym [43].
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 unexplored code, while less sophisticated strategies risk getting stuck (e.g., in loops). KLEE [8] and Mayhem [10] 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 the more recent systems running symbolic execution in concolic mode along with a fuzzer. Therefore, we use concolic mode for all systems, 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 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 [27].

For reasons unknown to us, there are symbolic execution engines that do not 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.

3.3 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 [43]. 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 this aspect of symbolic execution. We discuss this problem in more detail in Section 5.4.

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 [1].

4 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 [8] for source-based IR generation, S2E [11] for binary-based generation, angr [38] as a binary-based approach with a different IR, and Qsym [43] 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.

KLEE. Published in 2008, KLEE [8] is a well-known symbolic execution engine that is commonly used as a basis for further research [7, 11, 13, 14, 24, 33]. KLEE interprets LLVM bitcode, the

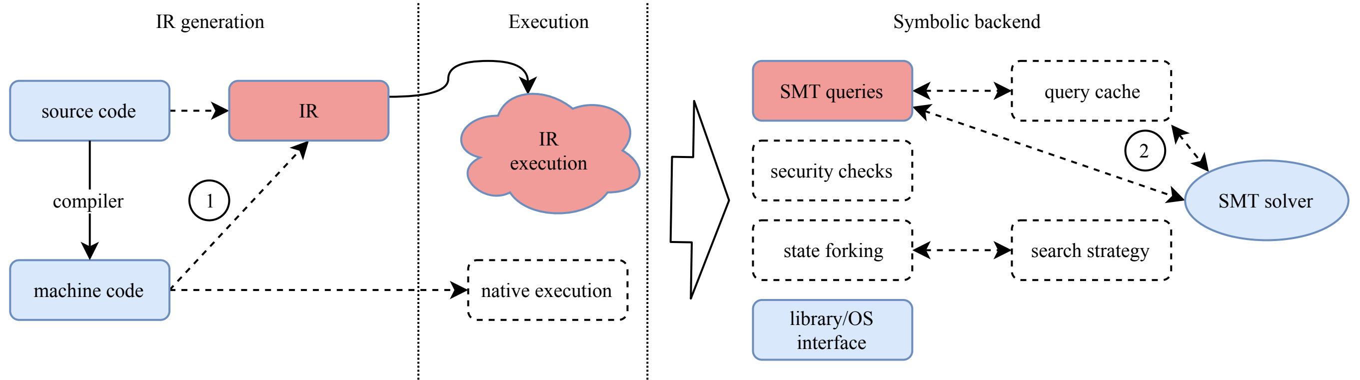

Figure 1: Overview of symbolic execution, showing our focus on IR, IR execution, and SMT queries. Numbers indicate orthogonal studies by (1) Kim et al. [21] and (2) Palikareva and Cadar [29] as well as Liu et al. [27]. Dashed elements are not always present. IR and machine code may be identical (e.g., in Qsym [43]).

Name	Version	IR	IR generator	Solver(s)	Programming language	Concrete execution
KLEE	`4efd7f6`	LLVM	Clang (from source)	Z3 (4.4.1), others	C++	None
S2E	2018-09-24	LLVM	QEMU + lifter	Z3 (4.7.1)	C, C++ (mostly)	QEMU with KVM
angr	7.8.8.1	VEX	libvex	Z3 (4.5.1)	Python	Unicorn
Qsym	`6f00c3d`	x86 machine code	n/a	Z3 (4.5.0)	C++ (mostly)	Directly on the CPU

Table 1: Comparison of design choices relevant to our study in the four symbolic execution engines that we analyze.

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 [6] by default but also supports Z3 [15], which we use as a common ground in our study. KLEE executes all user code at the IR level.

S2E. In order to address several perceived shortcomings of KLEE, Chipounov et al. proposed *Selective Symbolic Execution (S2E)* [11]. 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 [3], 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.

angr. Shoshitaishvili et al. created angr [38] 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 [28]. 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 [32]. The core emulator 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]

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 [43]. 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 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.

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

[2] The authors recommend executing angr in PyPy, a JIT-compiling implementation of Python; we use PyPy version 5.1.2.

5 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 motivated by the observations above:

(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.

5.1 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 find 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 [10, 37].
- 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 [34].

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 IR; 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 variables as possible by identifying design decisions that could introduce noise into our measurements (see Section 3) 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 binaries 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 binaries 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 [43]. The CGC suite contains a total of 131 different programs.

As discussed in Section 3.1, 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. [43], is the implicit 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 happens 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 make 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 and 26 LoC to Qsym (timing and query logging in both cases). 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 each use one core of an Intel Xeon Gold 6130 CPU. 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.

5.2 Benchmark size

Out of the 131 CGC programs, only 24 execute successfully in all four symbolic execution engines (see Table 2; Appendix C describes the applications). 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

	Qsym	S2E	angr	KLEE	all
Execution speed (Section 5.4)	70.2%	66.4%	75.6%	35.1%	18.3%
Query complexity (Section 5.5)	57.3%	74.8%	87.8%	38.9%	17.6%

Table 2: Percentage of CGC programs (out of 131) that we were able to use per experiment and symbolic execution engine. See Appendix C for more details.

of C/C++ across various libraries [35]; and this addresses a single limitation in a single tool. In general, the missing features typically depend on 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 [30] and by Xu et al. [42]. Finally, fairness requires us to base our comparison only on targets that are supported by *all* engines, which further restricts the test set.

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 paper attempts to take a step in that direction.

5.3 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 IR statement than IR derived from binaries.[3] 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 [19] 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

[3]There is the additional effect that some information is actually lost during compilation, such as buffer sizes [12]. 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 1).

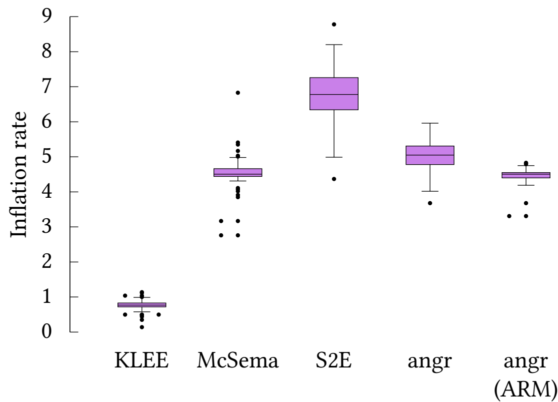

Figure 2: Inflation factor per IR generation mechanism, i.e., the number of generated IR instructions per machine-code instruction, across all tested programs (123 CGC and 106 coreutils binaries). The box encloses the second and third quartile of the data with a horizontal line marking the median. The whiskers include data points up to 1.5 times the interquartile range away; outliers beyond that point are depicted individually.

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 [16] 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 shows the results of our measurements, and Figure 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 the size of the generated translation artifacts. It is more meaningful when the target of the translation process is the same IR, which removes

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: 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 2. The CGC data set contains 123 programs, the coreutils suite 106 programs.

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 (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. The following experiments assess the properties of symbolic execution on such IR.

5.4 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 3.3. 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.

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

Table 4: Execution modes used by the symbolic execution engines in our study. For Qsym, the "IR" in symbolic mode is machine code.

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 use 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 4 shows the use of the various execution modes by the systems in our analysis. We are interested in the execution of IR, so 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 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 5.1. In terms of Table 4, we capture the last two columns, which contains any possible execution of IR. The result is a quantity that we call *execution rate*; it represents the number of instructions executed per unit of time. Figure 3 shows the rates we measured. 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 the inflation factors from Table 3.[4] In other words, we obtain a measure of execution speed that is comparable across different IR generation processes.

[4]See Appendix B for a visualization of the untranslated rates, expressed in IR instructions per time.

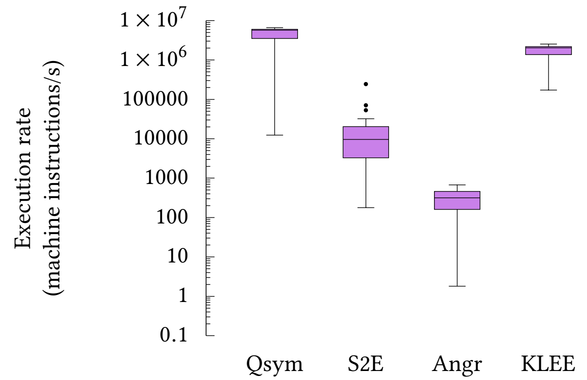

Figure 3: Execution speed of symbolically executed instructions, translated to the common basis of machine-code instructions, across 24 CGC programs. Higher rates mean faster execution.

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 5.3), 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++. See Appendix B for data supporting this hypothesis.

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 5.3)—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.

5.5 Query Complexity

Along with IR execution, SMT solving is one of the major workloads in symbolic execution [27, 29]. 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 1; it just reads five bytes from standard input, checks a number of conditions on the input and prints a result message. Listings 2 and 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 1: A simple program to demonstrate SMT queries.

```
1  #include <stdio.h>
2
3  int main(int argc, char* argv[]) {
4      char data[5];
5
6      for (int i = 0; i < 5; i++)
7          data[i] = getchar();
8
9      if (data[0] > 15 && data[1] == 32 &&
10         data[2] > 27 && data[2] < 100 &&
11         data[3] == 55 && data[4] == 123)
12         printf("Correct!\n");
13     else
14         printf("Try again...\n");
15
16     return 0;
17 }
```

Listing 2: Part of S2E's assertion for Listing 1. We use standard SMT-LIB syntax [2] for SMT queries.

```
1  (= (_ bv0 64)
2     (bvand
3      (bvadd
4       ;; 0xFFFFFFFFFFFFFFC9
5       (_ bv18446744073709551561 64)
6       ((_ zero_extend 56)
7        ((_ extract 7 0)
8         (bvor
9          (bvand
10          ((_ zero_extend 56)
11           (select stdin (_ bv3 32)))
12          ;; 0x00000000000000FF
13          (_ bv255 64))
14         ;; 0xFFFF88000AFDC000
15         (_ bv18446612132498620416 64)))))
16     (_ bv255 64)))
```

Listing 3: Part of KLEE's assertion for Listing 1.

```
1  (= (_ bv55 8)
2     ((_ extract 7 0)
3      ((_ zero_extend 24)
4       (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.

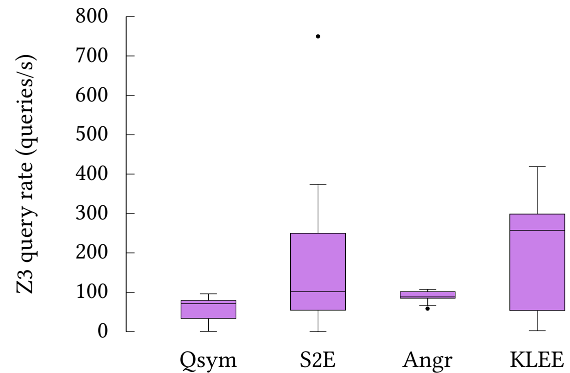

Figure 4: Comparison of the query rates for each system (using a common solver) as a proxy for query complexity, across 23 CGC programs. Higher rates indicate queries that are easier to solve. Note the differences in median.

We measure the time taken by Z3 to solve all the logged queries of successful executions as per Section 5.1. The four symbolic execution engines install different versions of Z3 (see Table 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 4 shows the *query rates* we measured, 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.

6 DISCUSSION

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

6.1 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 find 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 find 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.

6.2 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 1 would introduce variables 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 more feasible.

6.3 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 couldn't 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.

6.4 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.

7 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.

7.1 Symbolic Execution

Symbolic execution lies on a spectrum between more rigorous approaches, such as model checking [18, 31], and techniques that sacrifice soundness for practicality, such as fuzz testing [17]. Apart from the four symbolic execution engines that form the basis of our analysis, namely KLEE [8], S2E [11], angr [38] and Qsym [43], each representing a design category as described in Section 4, several others have been proposed and implemented. Manticore [40] is similar in focus to angr and implemented in Python as well but does not use any intermediate representation. Triton [36] is based on dynamic binary translation, like Qsym. Mayhem [10], based on BAP [5], is the winner of the DARPA CGC competition (but not freely available, and BAP alone does not support symbolic execution in recent versions). SAGE [20] is a closed-source system developed by Microsoft, following a concolic execution approach. Inception [14], 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 [33], support for floating-point arithmetic [13], parallel analysis [7], or state merging [24]. Recently, combining symbolic execution with fuzzing has been shown to hold great promise [39, 43].

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 [29] and the impact of incremental SMT solving [27]. A recent survey by Baldoni et al. [1] covers the general subject area of symbolic execution, and Xu et al. survey challenges of the field [41].

7.2 Intermediate Representations

There are a variety of intermediate representations. LLVM bitcode [25], employed by KLEE and S2E, was originally designed for use inside compilers. VEX [28], used by angr, targets binary instrumentation and was conceived for the Valgrind framework. Others, such as REIL [26] and BIL [5] have been developed specifically for security analysis. Kim et al. [21] 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.

8 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.

ACKNOWLEDGEMENTS

We are grateful to Insu Yun for detailed information on the evaluation of Qsym [43], which greatly simplified its setup for the purpose of our study. Moreover, we would like to thank Vitaly Chipounov for his help in debugging problems with S2E [11]. Experiments presented in this paper were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations (see https://www.grid5000.fr). This work was supported by the DAPCODS/IOTics ANR 2016 project (ANR-16-CE25-0015).

REFERENCES

[1] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. 2018. A survey of symbolic execution techniques. *ACM Computing Surveys (CSUR)* 51, 3 (2018), 50.
[2] Clark Barrett, Aaron Stump, and Cesare Tinelli. 2010. The SMT-LIB standard: Version 2.0. In *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, England)*, Vol. 13. 14.
[3] Fabrice Bellard. 2005. QEMU, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX Track*, Vol. 41. 46.
[4] Ella Bounimova, Patrice Godefroid, and David Molnar. 2013. Billions and Billions of Constraints: Whitebox Fuzz Testing in Production. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE '13)*. IEEE Press, Piscataway, NJ, USA, 122–131. http://dl.acm.org/citation.cfm?id=2486788.2486805
[5] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J Schwartz. 2011. BAP: A binary analysis platform. In *International Conference on Computer Aided Verification*. Springer, 463–469.
[6] Robert Brummayer and Armin Biere. 2009. 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, 174–177.
[7] Stefan Bucur, Vlad Ureche, Cristian Zamfir, and George Candea. 2011. Parallel symbolic execution for automated real-world software testing. In *Proceedings of the 6th ACM SIGOPS/EuroSys Conference on Computer Systems*. ACM, 183–198.
[8] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *OSDI*, Vol. 8. 209–224.
[9] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. 2008. EXE: automatically generating inputs of death. *ACM Transactions on Information and System Security (TISSEC)* 12, 2 (2008), 10.
[10] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. 2012. Unleashing mayhem on binary code. In *2012 IEEE Symposium on Security and Privacy*. IEEE, 380–394.
[11] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. 2011. S2E: A platform for in-vivo multi-path analysis of software systems. In *ACM SIGARCH Computer Architecture News*, Vol. 39. ACM, 265–278.
[12] Cristina Cifuentes and K. John Gough. 1995. Decompilation of binary programs. *Software: Practice and Experience* 25, 7 (1995), 811–829.
[13] Peter Collingbourne, Cristian Cadar, Paul H.J. Kelly, et al. 2011. Symbolic crosschecking of floating-point and SIMD code. In *European Conference on Computer Systems (EuroSys 2011)*.

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

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

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

[17] Joe W. Duran and Simeon Ntafos. 1981. A Report on Random Testing. In *Proceedings of the 5th International Conference on Software Engineering (ICSE '81)*. IEEE Press, Piscataway, NJ, USA, 179–183. http://dl.acm.org/citation.cfm?id=800078.802530

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

[19] Free Software Foundation. 2016. Coreutils - GNU core utilities. https://www.gnu.org/software/coreutils/. Accessed: 2019-02-04.

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

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

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

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

[24] Volodymyr Kuznetsov, Johannes Kinder, Stefan Bucur, and George Candea. 2012. Efficient state merging in symbolic execution. *Acm Sigplan Notices* 47, 6 (2012), 193–204.

[25] Chris Lattner and Vikram Adve. 2004. 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, 75.

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

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

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

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

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

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

[32] Nguyen Anh Quynh and Dang Hoang Vu. 2015. Unicorn – The ultimate CPU emulator. https://www.unicorn-engine.org/. Accessed: 2019-02-26.

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

[34] Eric F Rizzi, Sebastian Elbaum, and Matthew B. Dwyer. 2016. 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, 132–143.

[35] S2E issue tracker. 2019. Add support for symbolic MMX registers. https://github.com/S2E/s2e-env/issues/144. Accessed: 2019-06-04.

[36] Florent Saudel and Jonathan Salwan. 2015. Triton Dynamic Binary Analysis Framework. https://github.com/JonathanSalwan/Triton. Accessed: 2019-04-02.

[37] Edward J Schwartz, Thanassis Avgerinos, and David Brumley. 2010. 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, 317–331.

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

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

[40] Trail of Bits. 2017. Manticore: Symbolic execution for humans. https://blog.trailofbits.com/2017/04/27/manticore-symbolic-execution-for-humans/. Accessed: 2019-02-27.

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

[42] Hui Xu, Yangfan Zhou, Yu Kang, and Michael R. Lyu. 2017. 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, 181–188.

[43] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. 2018. QSYM: A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing. In *27th USENIX Security Symposium (USENIX Security 18)*. 745–761.

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

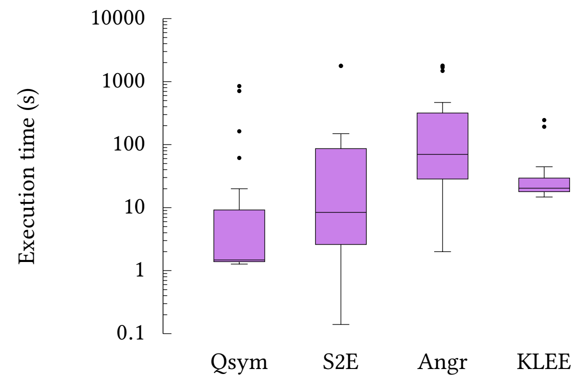

Figure 5: Absolute execution time during the measurement of execution speed.

A CODE AND DATASET AVAILABILITY

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

B ADDITIONAL MEASUREMENTS

This section shows measurement results that are complementary to those we present in Section 5. While they are not essential to our study, we assume that some readers will be interested in the additional data.

Figure 5 displays the absolute execution times of our measurements of execution speed in Section 5.4. In particular, we see that angr consumes an order of magnitude more time than S2E. Figure 6 shows the execution rates; however, here we express them in terms of each system's own IR instructions per time, i.e., before translating to the common basis of machine code instructions. Note that the relative order of the four systems is the same as in Figure 3.

Figure 7 visualizes the absolute number of queries generated by each system in our measurement of query complexity (see Section 5.5). We note that angr and KLEE tend to issue more queries than S2E and Qsym. However, we attribute the differences to the varying degrees of instrumentation in the implementations rather than the IR or its generation. For instance, KLEE performs bounds

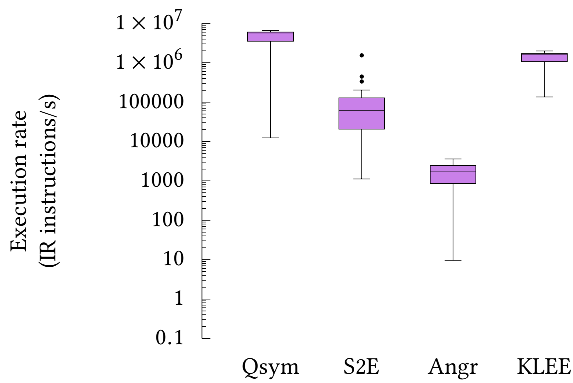

Figure 6: Execution speed of symbolically executed instructions. Higher rates mean faster execution.

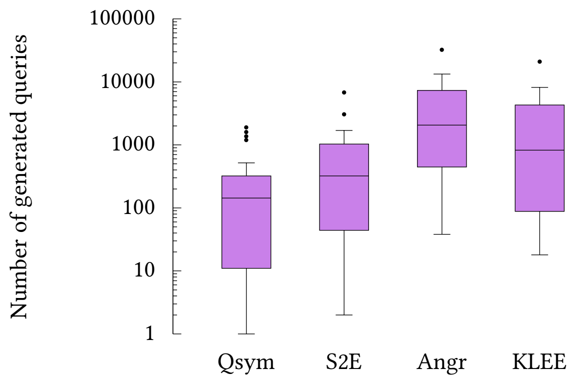

Figure 7: Absolute number of queries generated by the symbolic execution engines during our measurements of query complexity.

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 1).

C TESTED APPLICATIONS

Table 5 provides an overview of the CGC programs that we used for the experiments in Sections 5.4 and 5.5. When programs can be used successfully for the speed measurements in Section 5.4 but not for the assessment of query complexity in Section 5.5, the reason is often that the generated queries are so complex that the solver times out. (Recall that we do not set a timeout for individual queries when evaluating their complexity.)

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.

Name	Code size (LoC)	Used in Section 5.4	Used in Section 5.5	Description
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 5: Details of the CGC programs used in our measurements of execution speed and query complexity.