SymQEMU: Compilation-based symbolic execution for binaries

Sebastian Poeplau, Aurélien Francillon

▶ To cite this version:

Sebastian Poeplau, Aurélien Francillon. SymQEMU: Compilation-based symbolic execution for binaries. NDSS 2021, Network and Distributed System Security Symposium, Feb 2021, San Diego (virtuel), United States. 10.14722/NDSS.2021.24118 . hal-03556899

HAL Id: hal-03556899
https://hal.science/hal-03556899

Submitted on 4 Feb 2022

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

SymQEMU: Compilation-based symbolic execution for binaries

Sebastian Poeplau
EURECOM and Code Intelligence
sebastian.poeplau@eurecom.fr

Aurélien Francillon
EURECOM
aurelien.francillon@eurecom.fr

Abstract—**Symbolic execution is a powerful technique for software analysis and bug detection. Compilation-based symbolic execution is a recently proposed flavor that has been shown to improve the performance of symbolic execution significantly when source code is available. 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, modifying 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.

I. Introduction

Symbolic execution is becoming increasingly popular in program testing. Research over the past few decades has steadily improved the design and increased the performance of available implementations [2], [12]. Nowadays, symbolic execution has a reputation of being a highly effective yet expensive technique to explore programs. It is often combined with fuzz testing (so-called *hybrid fuzzing*), where the fuzzer leverages heuristics to explore relatively easy-to-reach paths quickly, while symbolic execution contributes test cases that reach the more difficult-to-explore parts of the target program [27], [28].

An important characteristic of symbolic execution systems is whether they require the *source code* of the program under test 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 [24], [29].
- 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 [28] 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 [6] 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 paper, 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 [3] can be combined with a very lightweight mechanism for symbolic execution: instead of 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-

Network and Distributed Systems Security (NDSS) Symposium 2021
21-25 February 2021, Virtual
ISBN 1-891562-66-5
https://dx.doi.org/10.14722/ndss.2021.23118
www.ndss-symposium.org

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 make SymQEMU publicly available to foster future research in the area.[1]

Note that the notion of compiling symbolic handling into target programs is also at the core of our previous work SymCC [20]. We showed that it outperforms other current approaches to symbolic execution. However, SymCC is only applicable when source code is available, and therefore does not support binary analysis. SymQEMU, in contrast, demonstrates how to achieve similar performance gains in a binary-only setting, respecting all the additional constraints inherent to that scenario (see Section II). For a more detailed comparison of SymQEMU and SymCC, refer to Section III-D.

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. Finally, we submitted SymQEMU to Google FuzzBench, a comparison framework for fuzzers; even though the test suite is not targeted at symbolic execution systems, SymQEMU outperformed all included fuzzers on 3 out of 21 targets.

In summary, we make 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. The source code of our implementation is freely available.
- We evaluate our system in Google FuzzBench, as well as on open-source and closed-source real-world software. The raw results, as well as data and scripts used in our evaluation, will be published with the paper.

The remainder of the paper is structured as follows: We first review symbolic execution in general, as well as the binary-only flavor in particular, and existing implementations thereof (Section II). Then we present design and implementation of SymQEMU (Section III) and evaluate it against the state of the art (Section IV). Finally, we discuss future work (Section V), place our contribution in the context of previous work (Section VI), and conclude (Section VII).

II. Background

In this section, we present symbolic execution in general before we examine the challenges of binary-only symbolic execution and review how state-of-the-art implementations address them. Finally we discuss SymCC, a source-based symbolic executor that was an important inspiration for our work.

[1]http://www.s3.eurecom.fr/tools/symbolic_execution/symqemu.html

A. Symbolic execution

The general goal of symbolic execution is to keep track of how intermediate values are computed during the execution of a target program. Usually, each intermediate value can be expressed as a formula in terms of the program input(s). Then, at any point during execution, the system can leverage those symbolic expressions 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.

In order to trace computations in the target program, symbolic execution systems need a certain understanding of the program's instruction set. Many current implementations translate the program to an *intermediate representation* [4], [6], [25]; typical examples of such representations are LLVM bitcode [14] and VEX [17]. The intermediate representation is subsequently executed symbolically; since the executor only needs to handle the intermediate language (usually consisting of a rather low number of instructions), the implementation can be relatively simple. Moreover, we found in previous work that queries derived from high-level representations of the program under test are easier to solve than those derived from low-level instruction sets like machine code [19].

However, translating programs to intermediate representations requires computational effort and introduces overhead in program execution; some implementations therefore choose to forego any form of translation and work on machine code directly [22], [28]. Apart from the performance benefits, skipping program translation helps robustness because concrete machine code can be executed even when the symbolic executor does not know how to interpret a given instruction. On the downside, specializing on the machine code of a particular processor architecture restricts the symbolic execution system to that platform. The alternative extreme—working directly on source code—is less common these days and obviously does not apply when only a binary is available.

B. Binary-only symbolic execution

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 [18], most implementations perform the translation on demand at run time [6], [25]. 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 particularly relevant for the embedded space, where a large variety of processor architectures is commonplace.

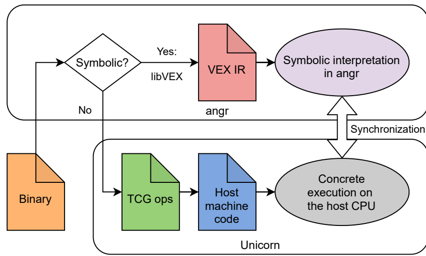


Fig. 1. Overview of angr: the target program is lifted to VEX IR and interpreted symbolically or executed concretely inside the Unicorn CPU emulator.

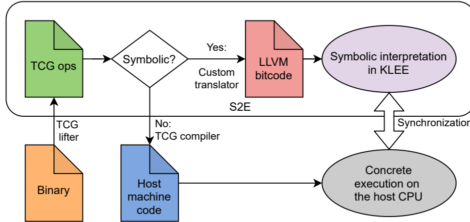


Fig. 2. Overview of S2E: the target program is lifted to TCG ops and then either translated to host machine code or lifted once more and executed symbolically in KLEE.

Translation-free symbolic executors thus face 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 [11]. This is in contrast to source-based symbolic execution, where intermediate representations can rather easily be obtained from the program's source code [4].

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

C. State-of-the-art solutions

Having presented the challenges of binary-only symbolic execution, we now describe three popular state-of-the-art implementations and study the design choices with which they address those problems.

1) Angr [25]: A "classic" translating symbolic executor. It reuses VEX, the intermediate language and translator of the Valgrind framework [17]. 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 [21], a fast CPU emulator based on QEMU [3]. Figure 1 illustrates the design.

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 [19] but very versatile.

2) S2E [6]: Created from the desire to extend the reach of the source-based symbolic-execution system KLEE [4] to the target program's dependencies and the operating-system kernel. To this end, S2E runs an entire operating system inside the emulator QEMU [3] and connects it to KLEE in order to execute relevant code symbolically (see Figure 2). The resulting system is rather complex, involving multiple translations of the program under test:

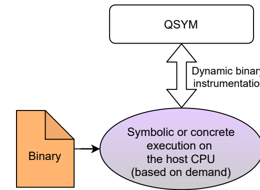


Fig. 3. Overview of QSYM: the target program is executed directly on the CPU while QSYM instruments it dynamically.

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 [14], 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.[2] However, the flexibility comes at a 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 [19]. 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.

3) QSYM [28]: 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 3). Concretely, it employs Intel Pin [15], 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.

D. SymCC

The recently presented symbolic executor SymCC [20], proposed by the same authors as the present publication, does

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

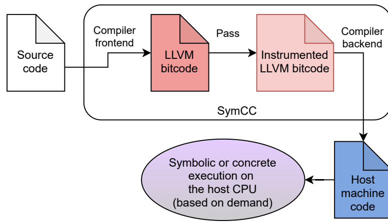


Fig. 4. Overview of SymCC: the source code of the target program is compiled to machine code; symbolic handling is injected at the level of LLVM bitcode in the compiler.

not work on binaries; however, SymQEMU draws inspiration from SymCC, so we briefly outline SymCC's design here. We refer interested readers to the original publication for details.

Our core observation when building SymCC was that most modern symbolic execution systems are interpreters. We proposed a compilation-based approach instead, showing that it increases execution performance as well as the overall exploration capability of the resulting system. SymCC hooks into compilers and instruments target code at compile time, injecting calls to a run-time support library. Symbolic execution thus becomes an integral part of the compiled program. Moreover, the analysis code benefits from compiler optimizations, and instrumentation work is not duplicated at every execution. Figure 4 illustrates the design.

SymCC's compilation-based approach fundamentally requires a compiler—it 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 the present paper is to demonstrate how compilation-based symbolic execution can, in fact, be made to work efficiently on binaries.

III. 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.

A. 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 II-C, 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 [19], whereas QSYM is very fast but intimately tied to the x86 platform [28].

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 [14] and static binary analysis [11]. 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 [18]), 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 II-C2). The second translation could be avoided if symbolic execution was implemented on the first intermediate representation.[3]
- 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 has recently shown that this is true of source-based symbolic execution [20], but its compiler-based design inherently requires source code and therefore doesn't apply to the binary-only use case (see Section II-D).

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 binary to an intermediate language in the first

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

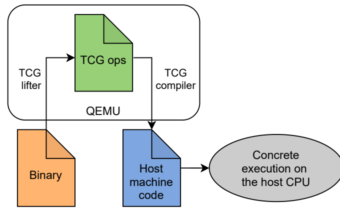


Fig. 5. Overview of regular QEMU: the target program is translated to TCG ops, which are subsequently compiled to machine code and executed on the host CPU.

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.

B. Implementation

We implemented SymQEMU on top of QEMU [3], 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" [17]; 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). The translated blocks are subsequently cached for performance reasons, so translation 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 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 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. 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

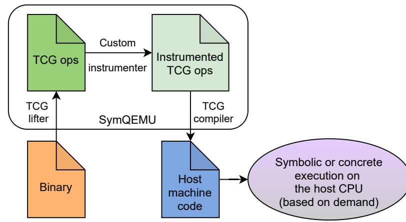


Fig. 6. Overview of SymQEMU: the target program is translated to TCG ops as in regular QEMU (see Figure 5), but before the compilation to host machine code we insert instructions to perform symbolic execution at run time.

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 SymQEMU could be extended to work with QEMU's full-system emulation if necessary (see Section V).

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 III-E).

C. 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 IV.)

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 [9]) on a more powerful machine. SymQEMU is well prepared for this use case: QEMU runs on all major host architectures.[4]

But what about guest architectures? SymQEMU leverages QEMU's TCG translators, which cover a wide range of processor types—the online documentation[5] 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 III-G). 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 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.

D. Comparison with previous designs

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

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 IV).

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 II-B): where SymCC instruments LLVM bitcode during (source-based) compilation, SymQEMU instruments TCG ops during dynamic binary translation. See Section III-F 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

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

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.

Table I summarizes the comparison. *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.

E. 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.

F. 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,[6] 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 [20].

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.

G. Shadow call stack

QSYM introduced the concept of context-sensitive basic-block pruning [28], a technique that suppresses symbolic

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

TABLE I. COMPARISON OF SYMQEMU WITH STATE-OF-THE-ART SYMBOLIC EXECUTION SYSTEMS.

Symbolic executor	Reference	Implementation language	Intermediate representation	Speed	Multiarch	Binary-only	Cross-architecture
angr	[25]	Python	VEX	✗	✓	✓	✓
S2E	[6]	C/C++	TCG & LLVM	✗	✓	✓	✓
QSYM	[28]	C++	none	✓	✗	✓	✗
SymCC	[20]	C++	LLVM	✓	✓	✗	✗
SymQEMU		C/C++	TCG	✓	✓	✓	✓

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.[7] 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.

TABLE II. SUMMARY OF THE FUZZBENCH RESULTS FOR 21 TARGETS. SYMQEMU RANKED FIRST ON 3 TARGETS, SECOND ON AVERAGE ACROSS ALL TARGETS, AND OUTPERFORMED PURE AFL ON 14.

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	✓	✗

IV. 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.

A. *FuzzBench*

Google announced FuzzBench in March 2020 as "a fully automated, open source, free service for evaluating fuzzers".[8] 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.[9] 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).

Figures 7, 8, 9 and 10 exemplify the outcome for two targets, and Table II summarizes the results; we show the ranking for all targets in Appendix B and the full report online.[10] 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;

[7] There is a *call* instruction in TCG, but it serves a different purpose.

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

[9] https://google.github.io/oss-fuzz/

[10] http://www.s3.eurecom.fr/tools/symbolic_execution/symqemu.html

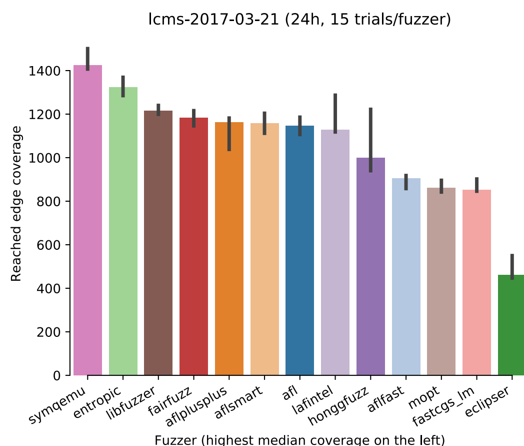


Fig. 7. Excerpt from the FuzzBench report: Ranking by median reached coverage for the FuzzBench target *lcms*. SymQEMU outperforms all other tools on this target.

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 IV-B). Finally, our rather crude integration simply dedicates a fixed share 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 [10]), could further improve the results (see Section VI-C).

B. 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 II-C2), and against QSYM because it is the fastest binary-only symbolic executor that we are aware of (see Section II-C3). We omitted angr (Section II-C1) from the comparison because preliminary experiments showed that its execution speed is significantly lower than that of the other tools; angr prioritizes versatility and ease of interactive use over raw speed [19]. Finally, we added raw AFL as a baseline, and we compared against SymCC (see Section II-D) 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, e.g., it can benefit from high-level code structures and compiler optimizations. Naturally, we can only evaluate against SymCC on open-source targets.

For our comparison, we performed hybrid fuzzing of a number of target programs and measured code coverage over time. We used AFL's notion of coverage for the same reason as in the evaluation of SymCC [20]: it is what drives AFL's exploration process. Following the recommendations by Klees et al. [13], 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 have previously used the three open-source tools for the evaluation of SymCC, 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 in QSYM and SymCC publications (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.[11] 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 gave AFL access to the target's source code when it was available and used QEMU mode otherwise.

[11] https://github.com/S2E/s2e/pull/20

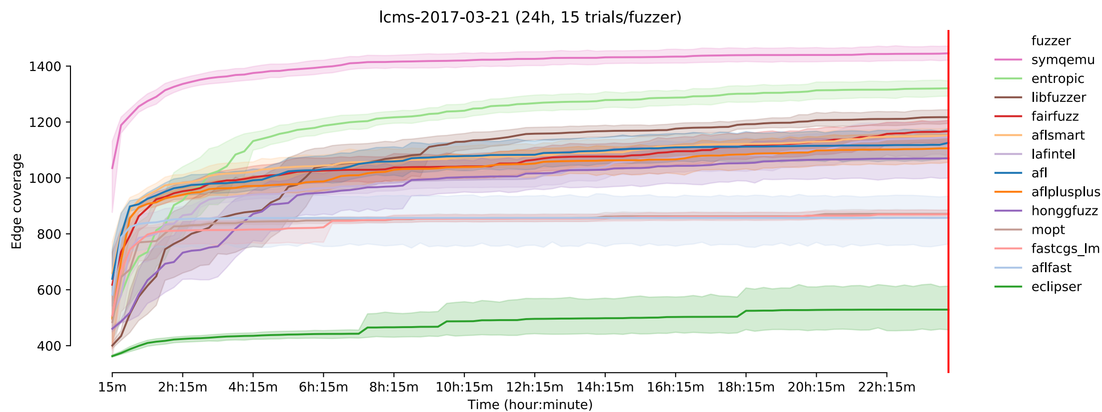


Fig. 8. Excerpt from the FuzzBench report: Mean coverage growth over time (and 95 % confidence intervals) for the FuzzBench target *lcms*. SymQEMU outperforms all other tools on this target.

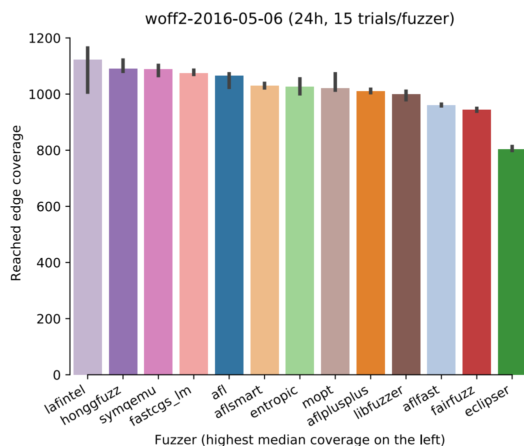


Fig. 9. Excerpt from the FuzzBench report: Ranking by median reached coverage for the FuzzBench target *woff2*. SymQEMU reaches 3rd rank on this target.

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 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 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 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.[12] This experiment shows that 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 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

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

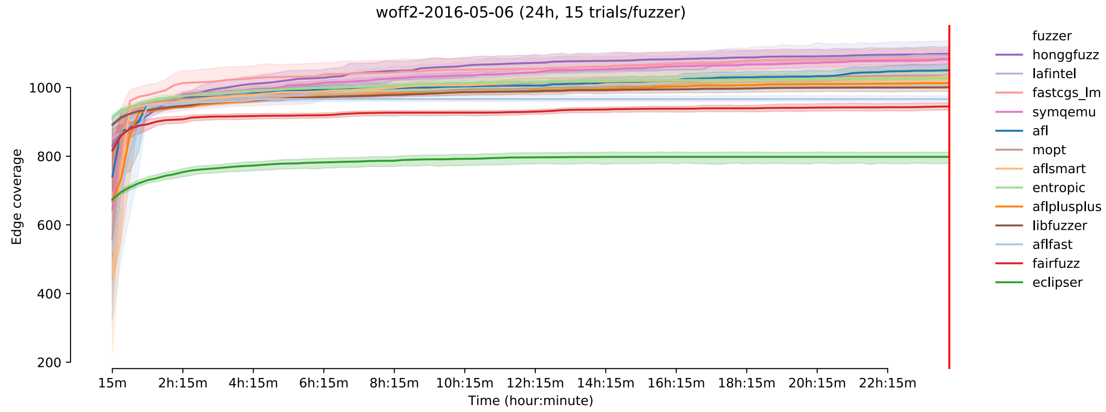


Fig. 10. Excerpt from the FuzzBench report: Mean coverage growth over time (and 95 % confidence intervals) for the FuzzBench target *woff2*. SymQEMU reaches 3rd rank on this target.

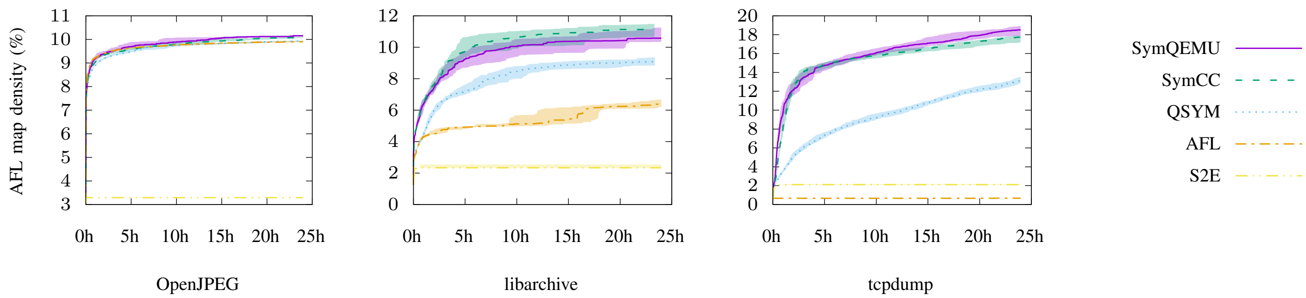


Fig. 11. Coverage over time on the open-source targets, expressed via the density of AFL's coverage map, showing median and 95 % confidence corridor. SymQEMU achieves higher coverage than all other systems with statistical significance (Mann-Whitney U, $p < 0.005$ two-tailed), except on libarchive, where there is no statistically significant difference with SymCC. Note, however, that SymCC requires the source code of the program under test.

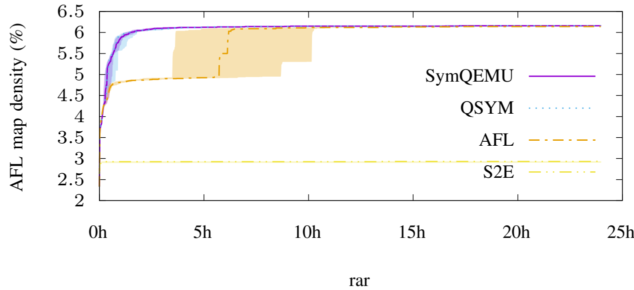


Fig. 12. Coverage over time on the closed-source rar program, expressed via the density of AFL's coverage map, showing median and 95 % confidence corridor. All tools except S2E converge towards the same coverage level, but SymQEMU reaches it faster than AFL and therefore requires less computing power per coverage. Moreover, its speed is similar to QSYM's, but QSYM cannot be easily ported from x86 to other targets.

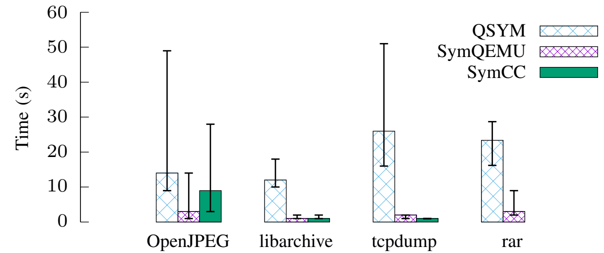


Fig. 13. Target execution times per symbolic executor and target program. Note that SymQEMU is faster than QSYM and at least as fast as the source-based SymCC. The notion of execution time is not applicable to S2E; SymCC cannot analyze rar because the source code is not available.

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.

C. 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 comparison, 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.

TABLE III. RESULTS OF OUR BENCHMARK COMPARISON ON FIXED INPUTS, VISUALIZED IN FIGURE 14.

		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	

In other words, we remove one variable from the comparison: the choice of paths to follow. Concretely, for each of the three open-source targets used in Section IV-B 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 14 and Table III 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 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 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 [19] that high-level intermediate representations lead to simpler SMT queries.

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 11 and 12, geometric mean) and 58 % faster execution in the benchmark experiment (Figure 14, geometric mean).

V. FUTURE WORK

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

A. 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 [16]; 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.

B. 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.

C. 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.[13] This approach is easy to implement but does not scale to large numbers of helpers.

[13] E.g., http://s2e.systems/docs/Plugins/Linux/FunctionModels.html.

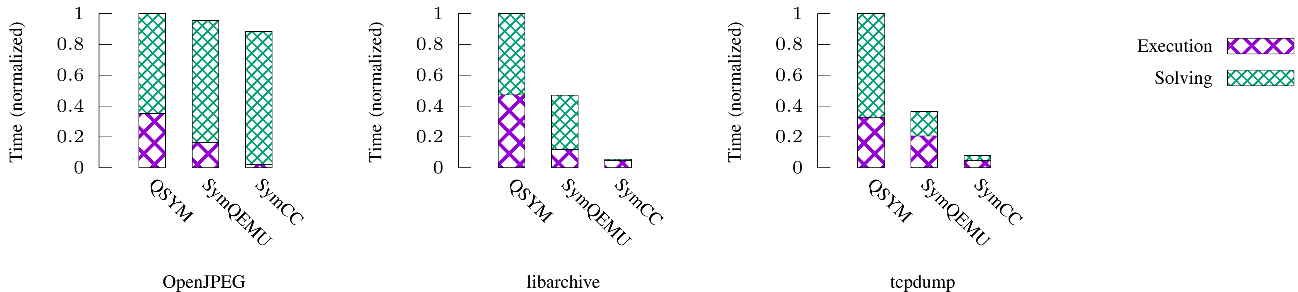


Fig. 14. Time spent in execution and SMT solving, respectively, averaged across concolic execution of a fixed set of test cases (1000 cases per target, chosen at random and analyzed in each of the three symbolic executors). Times are normalized to the total execution time of the slowest engine per target to show the differences in the overall amount of time required to complete the benchmark.

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.

VI. Related work

We now place SymQEMU in the context of previous work.

A. Binary-only symbolic execution

Angr [25], S2E [6], QSYM [28] and SymCC [20] have all been described in Section II, and we have compared them to SymQEMU in Section III-D. Mayhem [5] 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 [22] 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 [7] 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 [1] searches for correspondence between branch conditions and input bytes using a number of heuristics. SymQEMU, in contrast, implements "full" symbolic execution.

B. Run-time bug detection

Hybrid fuzzing relies on the fuzzer and sanitizers to detect bugs. Address sanitizer [23] 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 [8], a QEMU-based system that implements similar checks for binaries. There is a plethora of other sanitizers, often requiring source code [26]. 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.

C. Hybrid fuzzing

Driller [27] is a hybrid fuzzer based on angr, similar in concept to QSYM but slower because of its Python implementation and interpreter-based approach [19], [28]. 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 [10] 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. However, since such improvements are orthogonal to the speed of the symbolic executor (which is the main concern of SymQEMU), they are outside the scope of this paper.

VII. 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

The source code for SymQEMU is publicly available at http://www.s3.eurecom.fr/tools/symbolic_execution/

symqemu.html. At the same location, we also provide detailed instructions to reproduce our experiments, and we share the raw results of our own evaluation.

Acknowledgments

We thank the anonymous reviewers for their thoughtful feedback and suggestions which helped us to increase the quality of the paper. This work has been supported partly by the DAPCODS/IOTics ANR 2016 project (ANR-16-CE25-0015) and partly by the Defense Advanced Research Projects Agency (DARPA) under agreement number FA875019C0003.

References

[1] C. Aschermann, S. Schumilo, T. Blazytko, R. Gawlik, and T. Holz, "Redqueen: Fuzzing with input-to-state correspondence." in *Network and Distributed System Security Symposium (NDSS)*, vol. 19, 2019, pp. 1–15.

[2] R. Baldoni, E. Coppa, D. C. D'Elia, C. Demetrescu, and I. Finocchi, "A survey of symbolic execution techniques," *ACM Computing Surveys (CSUR)*, vol. 51, no. 3, p. 50, 2018.

[3] F. Bellard, "QEMU, a fast and portable dynamic translator," in *USENIX Annual Technical Conference, FREENIX Track*, vol. 41, 2005, p. 46.

[4] C. Cadar, D. Dunbar, and D. R. Engler, "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *OSDI*, vol. 8, 2008, pp. 209–224.

[5] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley, "Unleashing Mayhem on binary code," in *2012 IEEE Symposium on Security and Privacy*. IEEE, 2012, pp. 380–394.

[6] V. Chipounov, V. Kuznetsov, and G. Candea, "S2E: A platform for in-vivo multi-path analysis of software systems," in *ACM SIGARCH Computer Architecture News*, vol. 39, no. 1. ACM, 2011, pp. 265–278.

[7] J. Choi, J. Jang, C. Han, and S. K. Cha, "Grey-box concolic testing on binary code," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 736–747.

[8] A. Fioraldi, D. C. D'Elia, and L. Querzoni, "Fuzzing binaries for memory safety errors with QASan." [Online]. Available: https://andreafioraldi.github.io/assets/qasan-secdev20.pdf

[9] E. Gustafson, M. Muench, C. Spensky, N. Redini, A. Machiry, Y. Fratantonio, D. Balzarotti, A. Francillon, Y. R. Choe, C. Kruegel, and G. Vigna, "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, Sep. 2019, pp. 135–150. [Online]. Available: https://www.usenix.org/conference/raid2019/presentation/gustafson

[10] 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. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/SP40000.2020.00063

[11] S. Kim, M. Faerevaag, M. Jung, S. Jung, D. Oh, J. Lee, and S. K. 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.

[12] J. C. King, "Symbolic execution and program testing," *Communications of the ACM*, vol. 19, no. 7, pp. 385–394, 1976.

[13] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, "Evaluating fuzz testing," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 2123–2138.

[14] C. Lattner and V. 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.

[15] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," in *Acm sigplan notices*, vol. 40, no. 6. ACM, 2005, pp. 190–200.

[16] M. Muench, J. Stijohann, F. Kargl, A. Francillon, and D. Balzarotti, "What you corrupt is not what you crash: Challenges in fuzzing embedded devices," in *Network and Distributed System Security Symposium (NDSS)*, 2018.

[17] N. Nethercote and J. Seward, "Valgrind: A framework for heavyweight dynamic binary instrumentation," in *ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI 2007)*, vol. 42, no. 6. ACM, 2007, pp. 89–100.

[18] C. Pang, R. Yu, Y. Chen, E. Koskinen, G. Portokalidis, B. Mao, and J. Xu, "SoK: All you ever wanted to know about x86/x64 binary disassembly but were afraid to ask," *arXiv preprint arXiv:2007.14266*, 2020.

[19] S. Poeplau and A. 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.

[20] ——, "Symbolic execution with SymCC: Don't interpret, compile!" in *29th USENIX Security Symposium (USENIX Security 20)*. Boston, MA: USENIX Association, 2020. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/poeplau

[21] N. A. Quynh and D. H. Vu, "Unicorn – the ultimate CPU emulator," https://www.unicorn-engine.org/, 2015.

[22] F. Saudel and J. Salwan, "Triton: A dynamic symbolic execution framework," in *Symposium sur la sécurité des technologies de l'information et des communications, SSTIC, Rennes, France, June 3-5 2015*. SSTIC, 2015, pp. 31–54.

[23] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "AddressSanitizer: A fast address sanity checker," in *USENIX Annual Technical Conference (USENIX ATC 12)*, 2012, pp. 309–318.

[24] Y. Shoshitaishvili, R. Wang, C. Hauser, C. Kruegel, and G. Vigna, "Firmalice – automatic detection of authentication bypass vulnerabilities in binary firmware," in *Network and Distributed System Security Symposium (NDSS)*, 2015.

[25] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel *et al.*, "Sok: (state of) the art of war: Offensive techniques in binary analysis," in *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2016, pp. 138–157.

[26] D. Song, J. Lettner, P. Rajasekaran, Y. Na, S. Volckaert, P. Larsen, and M. Franz, "SoK: Sanitizing for security," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 1275–1295.

[27] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Driller: Augmenting fuzzing through selective symbolic execution," in *Network and Distributed System Security Symposium (NDSS)*, vol. 16, 2016, pp. 1–16.

[28] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim, "QSYM: A practical concolic execution engine tailored for hybrid fuzzing," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018, pp. 745–761.

[29] J. Zaddach, L. Bruno, A. Francillon, and D. 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.

Appendix A
S2E Resource Consumption

We encountered a few challenges related to resource consumption when setting up S2E for our experiments (see Section IV-B). While they are not essential to the discussion, we still think they are interesting to document.

A. Parallel S2E

S2E has a parallel mode, in which it starts multiple processes and assigns each a dedicated portion of the state

tree.[14] 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. 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.

B. 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.)

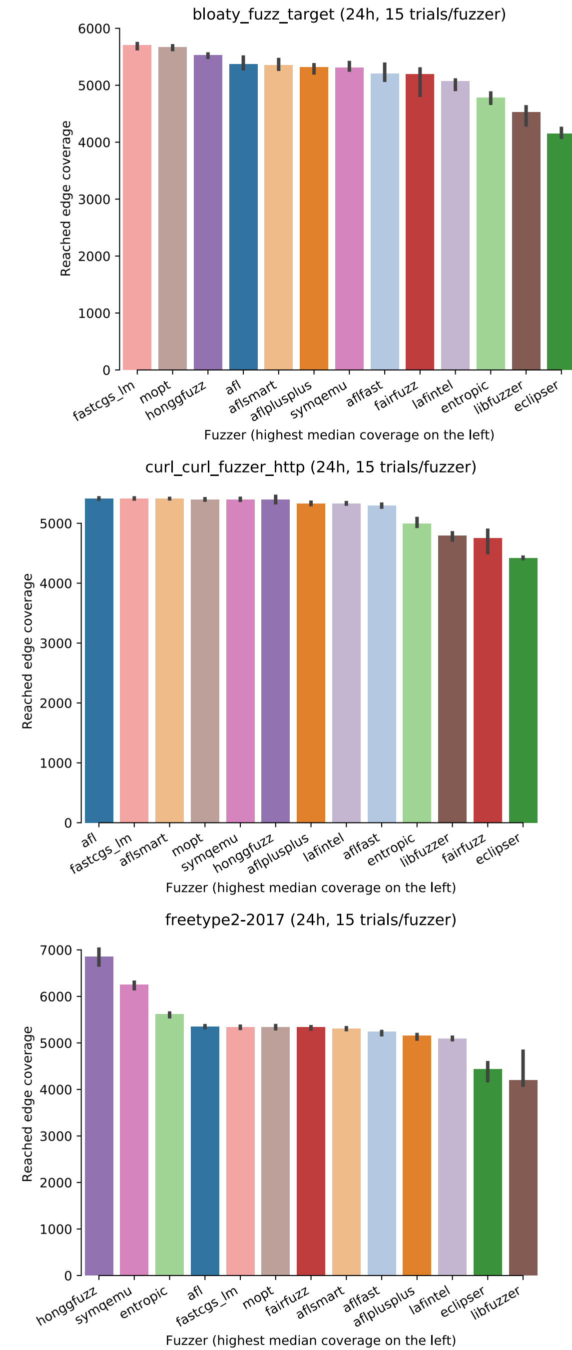
S2E provides the *ResourceMonitor* plugin for such cases.[15] Its task is to monitor memory consumption (with the limit defined via a cgroup) and prevent further forking or terminate execution states as consumption approaches the limit. Unfortunately, in our experiments, the plugin did not reduce memory 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's threshold (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.

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 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.

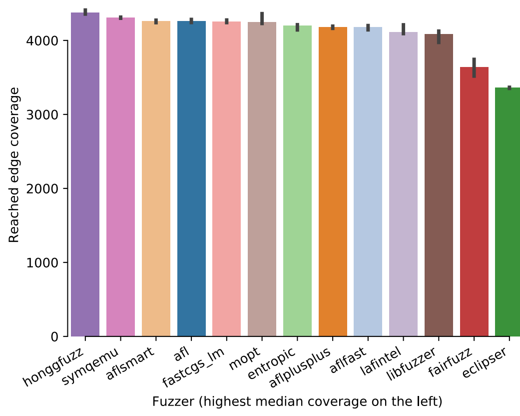
APPENDIX B
FUZZBENCH REPORT

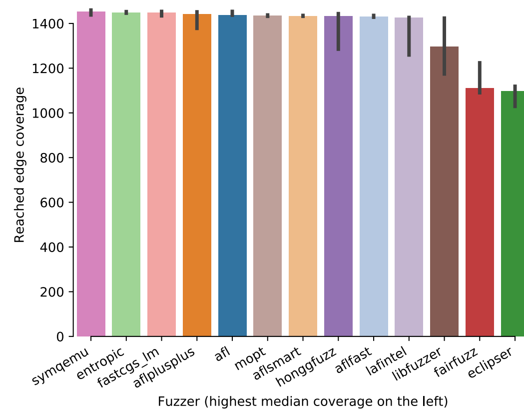
The figures below show the respective ranking of the fuzzers and our SymQEMU/AFL hybrid fuzzer on the 21 FuzzBench targets (see Section IV-A). We cannot include the full report for space reasons, but you can find the report rendered on our website;[16] in addition to the mere rankings, it shows coverage over time, statistical significance and coverage distribution across the different trials for each target.

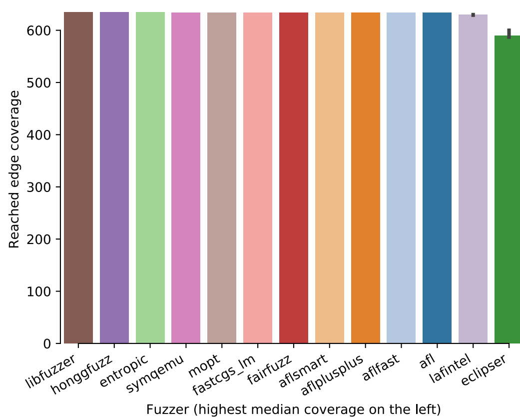
According to the authors of the FuzzBench suite, the low performance of all but five fuzzers on the libpcap target is due to a deficiency in AFL's code instrumentation, which all the low-performing fuzzers are based on. Similarly, SymQEMU depends on the fuzzer to identify promising test cases, so when the fuzzer's instrumentation fails it cannot make progress either.

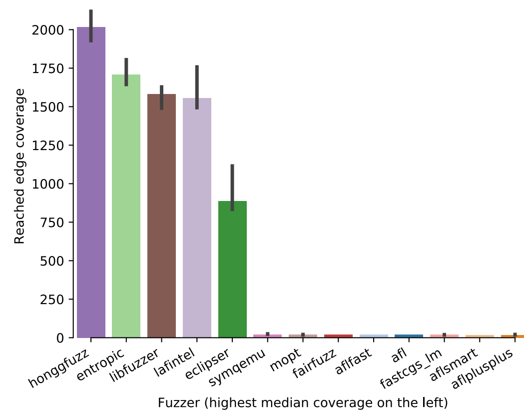


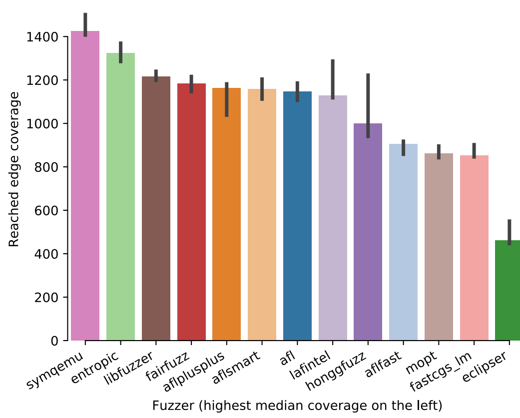
[14] http://s2e.systems/docs/Howtos/Parallel.html
[15] http://s2e.systems/docs/FAQ.html#how-to-keep-memory-usage-low
[16] http://www.s3.eurecom.fr/tools/symbolic_execution/symqemu.html

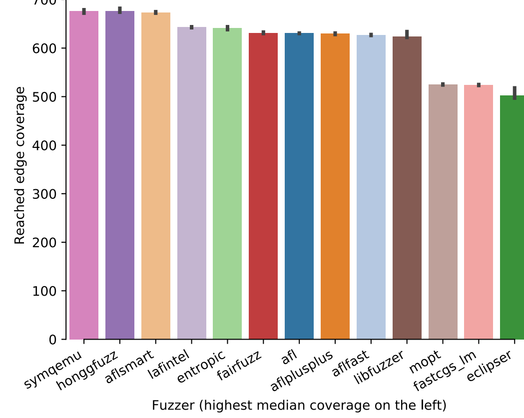

harfbuzz-1.3.2 (24h, 15 trials/fuzzer)
Reached edge coverage
honggfuzz
symqemu
aflsmart
afl
fastcgs_lm
mopt
entropic
aflplusplus
aflfast
lafintel
libfuzzer
fairfuzz
eclipser
Fuzzer (highest median coverage on the left)

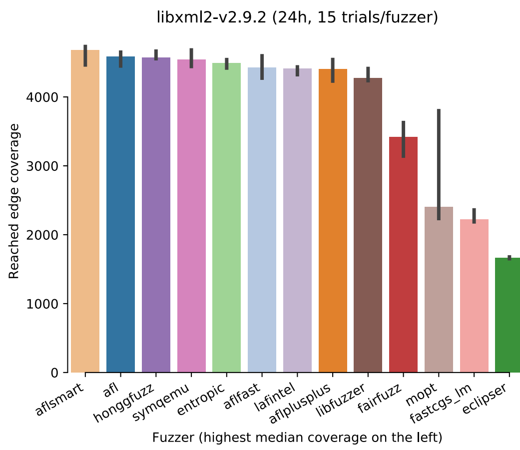

libjpeg-turbo-07-2017 (24h, 15 trials/fuzzer)
Reached edge coverage
symqemu
entropic
fastcgs_lm
aflplusplus
afl
mopt
aflsmart
honggfuzz
aflfast
lafintel
libfuzzer
fairfuzz
eclipser
Fuzzer (highest median coverage on the left)

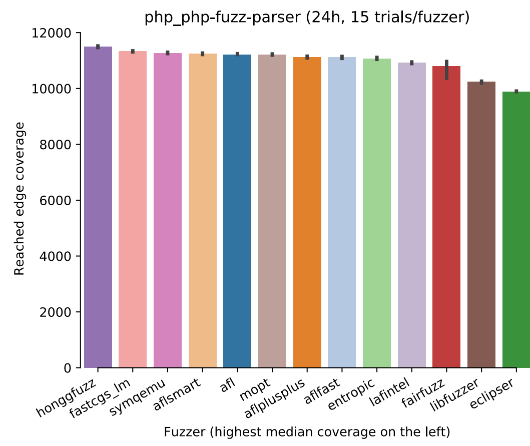

jsoncpp_jsoncpp_fuzzer (24h, 14 trials/fuzzer)
Reached edge coverage
libfuzzer
honggfuzz
entropic
symqemu
mopt
fastcgs_lm
fairfuzz
aflsmart
aflplusplus
aflfast
afl
lafintel
eclipser
Fuzzer (highest median coverage on the left)

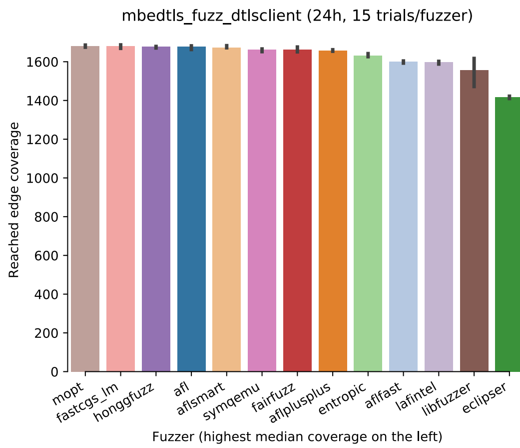

libpcap_fuzz_both (24h, 15 trials/fuzzer)
Reached edge coverage
honggfuzz
entropic
libfuzzer
lafintel
eclipser
symqemu
mopt
fairfuzz
aflfast
afl
fastcgs_lm
aflsmart
aflplusplus
Fuzzer (highest median coverage on the left)

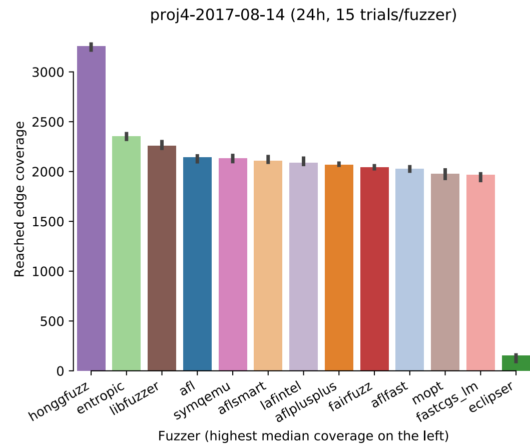

lcms-2017-03-21 (24h, 15 trials/fuzzer)
Reached edge coverage
symqemu
entropic
libfuzzer
fairfuzz
aflplusplus
aflsmart
afl
lafintel
honggfuzz
aflfast
mopt
fastcgs_lm
eclipser
Fuzzer (highest median coverage on the left)

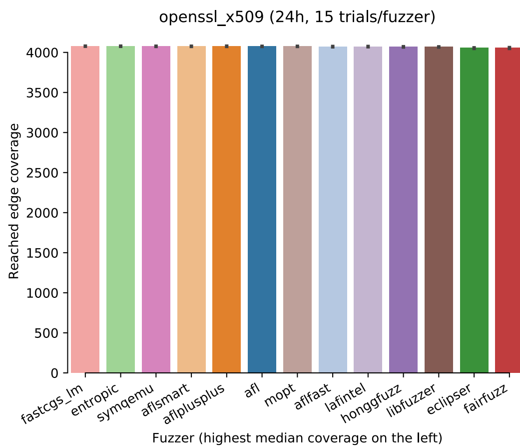

libpng-1.2.56 (24h, 15 trials/fuzzer)
Reached edge coverage
symqemu
honggfuzz
aflsmart
lafintel
entropic
fairfuzz
afl
aflplusplus
aflfast
libfuzzer
mopt
fastcgs_lm
eclipser
Fuzzer (highest median coverage on the left)

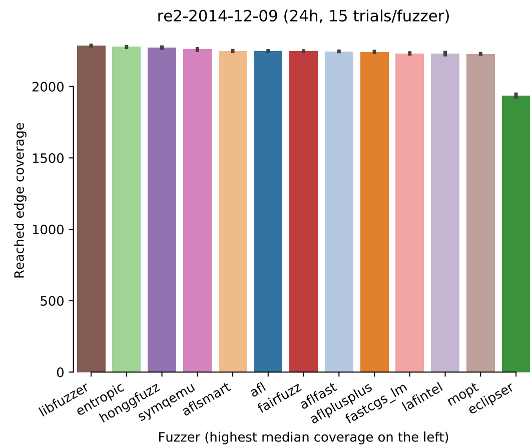

libxml2-v2.9.2 (24h, 15 trials/fuzzer)
Reached edge coverage
aflsmart
afl
honggfuzz
symqemu
entropic
aflfast
lafintel
aflplusplus
libfuzzer
fairfuzz
mopt
fastcgs_lm
eclipser
Fuzzer (highest median coverage on the left)

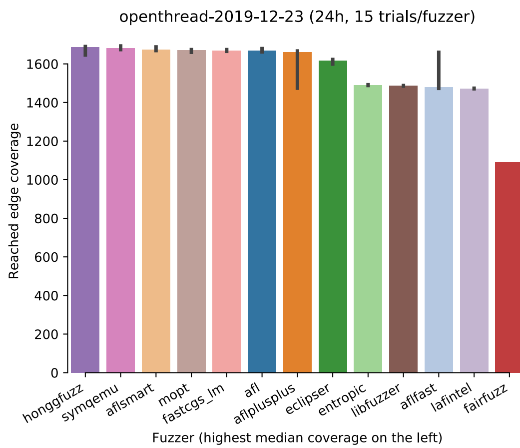

php_php-fuzz-parser (24h, 15 trials/fuzzer)
Reached edge coverage
honggfuzz
fastcgs_lm
symqemu
aflsmart
afl
mopt
aflplusplus
aflfast
entropic
lafintel
fairfuzz
libfuzzer
eclipser
Fuzzer (highest median coverage on the left)

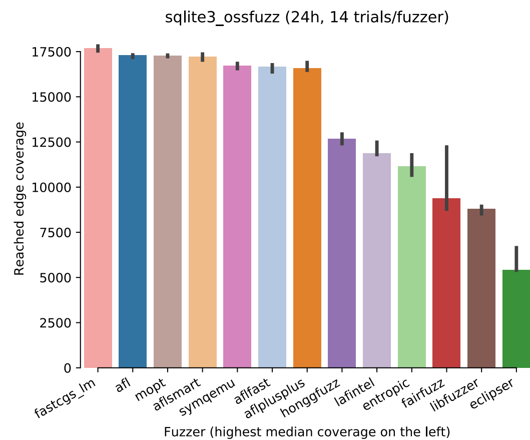

mbedtls_fuzz_dtlsclient (24h, 15 trials/fuzzer)
Reached edge coverage
mopt
fastcgs_lm
honggfuzz
afl
aflsmart
symqemu
fairfuzz
aflplusplus
entropic
aflfast
lafintel
libfuzzer
eclipser
Fuzzer (highest median coverage on the left)

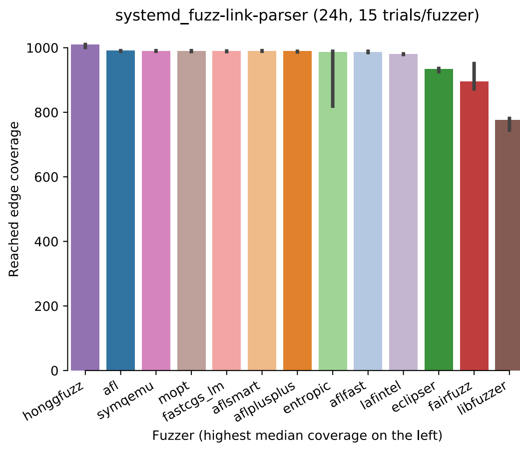

proj4-2017-08-14 (24h, 15 trials/fuzzer)
Reached edge coverage
honggfuzz
entropic
libfuzzer
afl
symqemu
aflsmart
lafintel
aflplusplus
fairfuzz
aflfast
mopt
fastcgs_lm
eclipser
Fuzzer (highest median coverage on the left)

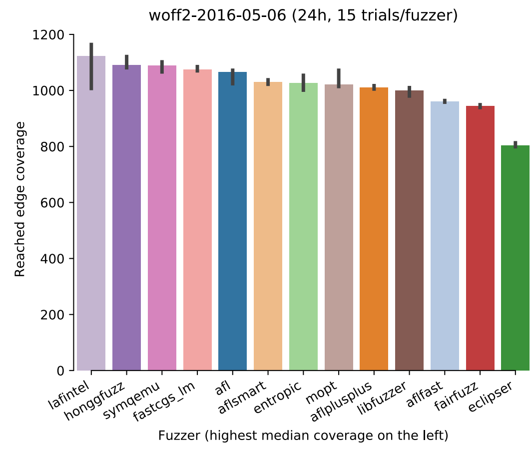

openssl_x509 (24h, 15 trials/fuzzer)
Reached edge coverage
fastcgs_lm
entropic
symqemu
aflsmart
aflplusplus
afl
mopt
aflfast
lafintel
honggfuzz
libfuzzer
eclipser
fairfuzz
Fuzzer (highest median coverage on the left)

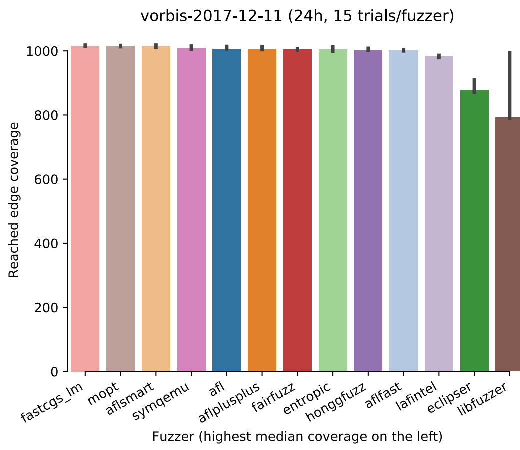

re2-2014-12-09 (24h, 15 trials/fuzzer)
Reached edge coverage
libfuzzer
entropic
honggfuzz
symqemu
aflsmart
afl
fairfuzz
aflfast
aflplusplus
fastcgs_lm
lafintel
mopt
eclipser
Fuzzer (highest median coverage on the left)

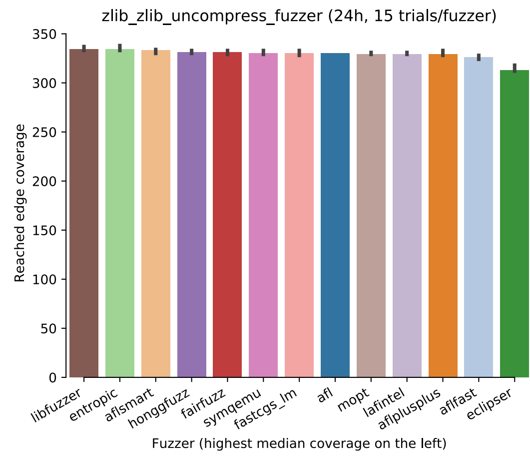

openthread-2019-12-23 (24h, 15 trials/fuzzer)
Reached edge coverage
honggfuzz
symqemu
aflsmart
mopt
fastcgs_lm
afl
aflplusplus
eclipser
entropic
libfuzzer
aflfast
lafintel
fairfuzz
Fuzzer (highest median coverage on the left)


sqlite3_ossfuzz (24h, 14 trials/fuzzer)
Reached edge coverage
fastcgs_lm
afl
mopt
aflsmart
symqemu
aflfast
aflplusplus
honggfuzz
lafintel
entropic
fairfuzz
libfuzzer
eclipser
Fuzzer (highest median coverage on the left)


systemd_fuzz-link-parser (24h, 15 trials/fuzzer)
Reached edge coverage
0
200
400
600
800
1000
honggfuzz
afl
symqemu
mopt
fastcgs_lm
aflsmart
aflplusplus
entropic
aflfast
lafintel
eclipser
fairfuzz
libfuzzer
Fuzzer (highest median coverage on the left)


woff2-2016-05-06 (24h, 15 trials/fuzzer)
Reached edge coverage
0
200
400
600
800
1000
1200
lafintel
honggfuzz
symqemu
fastcgs_lm
afl
aflsmart
entropic
mopt
aflplusplus
libfuzzer
aflfast
fairfuzz
eclipser
Fuzzer (highest median coverage on the left)


vorbis-2017-12-11 (24h, 15 trials/fuzzer)
Reached edge coverage
0
200
400
600
800
1000
fastcgs_lm
mopt
aflsmart
symqemu
afl
aflplusplus
fairfuzz
entropic
honggfuzz
aflfast
lafintel
eclipser
libfuzzer
Fuzzer (highest median coverage on the left)


zlib_zlib_uncompress_fuzzer (24h, 15 trials/fuzzer)
Reached edge coverage
0
50
100
150
200
250
300
350
libfuzzer
entropic
aflsmart
honggfuzz
fairfuzz
symqemu
fastcgs_lm
afl
mopt
lafintel
aflplusplus
aflfast
eclipser
Fuzzer (highest median coverage on the left)