Symbolic execution with SymCC: Don't interpret, compile!

Sebastian Poeplau and Aurélien Francillon, *EURECOM*

https://www.usenix.org/conference/usenixsecurity20/presentation/poeplau

This paper is included in the Proceedings of the 29th USENIX Security Symposium.

August 12–14, 2020

978-1-939133-17-5

Open access to the Proceedings of the 29th USENIX Security Symposium is sponsored by USENIX.

Symbolic execution with SymCC: Don't interpret, compile!

Sebastian Poeplau
EURECOM

Aurélien Francillon
EURECOM

Abstract

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

1 Introduction

Symbolic execution was conceived more than 40 years ago to aid in software testing [22]. While it was rather impractical initially, great advances in the field of computer-aided reasoning, in particular SAT and SMT solving, led to the first more or less practical implementations in the early 2000s [5, 6]. Since then, symbolic execution has been the subject of much research from both the software security and the verification communities [9, 37, 39, 45], and the technique has established its place in vulnerability search and program testing. In the 2016 DARPA Cyber Grand Challenge, a competition in automated vulnerability finding, exploiting and fixing, symbolic execution was an integral part in the approaches of all three winning teams [7, 30, 37].

Despite the increase in popularity, performance has remained a core challenge for symbolic execution. Slow processing means less code executed and tested per time, and therefore fewer bugs detected per invested resources. Several challenges are commonly identified, one of which is slow code execution: Yun et al. have recently provided extensive evidence that the execution component is a major bottleneck in modern implementations of symbolic execution [45]. We propose an alternative execution method and show that it leads to considerably faster symbolic execution and ultimately to better program coverage and more bugs discovered.

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

Interpretation is, in general, less efficient than compilation because it performs work at each execution that a compiler has to do only a single time [20, 44]. Our core idea is thus to apply "compilation instead of interpretation" to symbolic execution in order to achieve better performance. But what does compilation mean in the context of symbolic execution? In programming languages, it is the process of replacing instructions of the source language with sequences of machine code that perform equivalent actions. So, in order to apply the same idea to symbolic execution, we *embed* the symbolic processing into the target program. The end result is a binary that executes without the need for an external interpreter; it performs the same actions as the target program but additionally keeps track of symbolic expressions. This technique enables it to perform any symbolic reasoning that is conventionally applied by the interpreter, while retaining the speed of a compiled program.

Interestingly, a similar approach was used in early implementations of symbolic execution: DART [16], CUTE [35]

and EXE [6] instrument the program under test at the level of C source code. In comparison with our approach, however, they suffer from two essential problems:

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

2. The requirement to handle a full programming language makes the implementation very complex [16]; the approach may be viable for C but is likely to fail for larger languages like C++. Our compiler-based technique only has to handle the compiler's intermediate representation, which is a significantly smaller language.

The differences are discussed in more detail in Section 7.

We present an implementation of our idea, called SYMCC, on top of the LLVM framework. It takes the unmodified LLVM bitcode of a program under test and compiles symbolic execution capabilities right into the binary. At each branch point in the program, the "symbolized" binary will generate an input that deviates from the current execution path. In other words, SYMCC produces binaries that perform concolic execution, a flavor of symbolic execution that does not follow multiple execution paths at the same time but instead relies on an external entity (such as a fuzzer) to prioritize test cases and orchestrate execution (see Section 2 for details).

In the most common case, SYMCC replaces the normal compiler and compiles the C or C++ source code of the program under test into an instrumented binary.[1] As such, SYMCC is designed to analyze programs for which the source code (or at least LLVM bitcode) is available, for example during development as part of the secure development life cycle. It can, however, handle binary-only libraries and inline assembly gracefully. We discuss this aspect in more detail in Section 6.3. Appendix A demonstrates a typical user interaction with SYMCC.

In this paper, we first elaborate on our idea of compilation-based symbolic execution (Section 3). We then present SYMCC in detail (Section 4) and compare its performance with state-of-the-art implementations (Section 5), showing that it is orders of magnitude faster in benchmarks and that this speed advantage translates to better bug-finding capabilities in real-world software. Finally, we discuss the applicability of our novel technique and possible directions for future work (Section 6), and place the work in the context of prior research (Section 7).

In summary, we make the following contributions:

1. We propose compilation-based symbolic execution, a technique that provides significantly higher performance than current approaches while maintaining low complexity.

2. We present SYMCC, our open-source implementation on top of the LLVM framework.

3. We evaluate SYMCC against state-of-the-art symbolic execution engines and show that it provides benefits in the analysis of real-world software, leading to the discovery of two critical vulnerabilities in OpenJPEG.

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

2 Background

Before we describe compilation-based symbolic execution in detail, this section summarizes some relevant background information.

2.1 Symbolic execution

At its core, every implementation of symbolic execution is constructed from a set of basic building blocks (see Figure 1):

Execution The program under test is executed, and the system produces symbolic expressions representing the computations. These expressions are the essential asset for reasoning about the program. For our purposes, we distinguish between IR-based and IR-less execution, which are discussed in the subsequent two sections.

Symbolic backend The sole purpose of describing computations symbolically is to reason about them, e.g., to generate new program inputs that trigger a certain security vulnerability. The symbolic backend comprises the components that are involved in the reasoning process. Typically, implementations use an SMT solver, possibly enhanced by pre-processing techniques. For example, KLEE [5] employs elaborate caching mechanisms to minimize the number of solver queries, and QSYM [45] removes all irrelevant information from queries to reduce the load on the solver.

Forking and scheduling Some implementations of symbolic execution execute the target program only a single time, possibly along the path dictated by a given program input, and generate new program inputs based on that single execution. The new inputs are usually fed back into the system or passed to a concurrently running fuzzer. This approach, often referred to as *concolic execution*, is followed by SAGE [17], Driller [39] and QSYM [45], among others. On the other hand, several other implementations contain additional facilities to manage multiple executions of the program under test along different paths. Typically, they "fork" the execution at

[1] Support for additional source languages can be added with little effort; see Section 4.6.

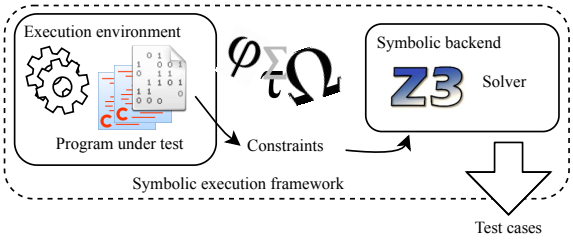

Figure 1: The building blocks of symbolic execution. The entire system may be encapsulated in a component that handles forking and scheduling.

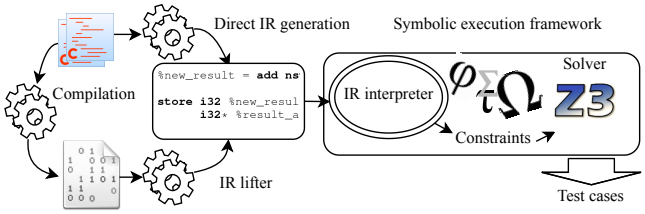

Figure 2: IR-based symbolic execution interprets IR and interacts with the symbolic backend at the same time.

branch points in the program (in order to avoid having to re-execute from the start with a new input); a scheduler usually orchestrates the different execution states and prioritizes them according to some search strategy. For example, KLEE [5], Mayhem [7] and angr [37] follow this approach.

The problem of *path explosion*, a term referring to system overload caused by too many possible paths of execution, is much more prevalent in this latter group of symbolic execution systems: A forking system needs to manage a considerable amount of information per execution state, whereas concolic executors simply generate a new program input, write it to disk, and "forget about it". Mayhem [7] implements a hybrid approach by forking while enough memory is available and persisting states to disk otherwise. For SYMCC, we decided to follow the concolic approach because we think that it allows for higher execution speeds and a simpler implementation.

The three building blocks—execution, symbolic backend, and forking/scheduling— are conceptually orthogonal to each other (with some technical dependencies between execution and forking), even if implementations sometimes lack a clear distinction. Our work focuses exclusively on improving the execution component, while we reuse the work of Yun et al. [45] for the symbolic backend.

We now examine the two prevalent flavors of the execution component in present implementations of symbolic execution.

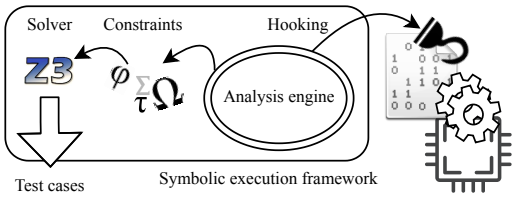

Figure 3: IR-less symbolic execution attaches to the machine code executing on the CPU and instruments it at run time.

2.2 IR-based symbolic execution

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

IR-based symbolic execution first needs to transform the program under analysis into IR. KLEE [5], for example, works on LLVM bitcode and uses the `clang` compiler to generate it from source code; S2E [9] also interprets LLVM bitcode but generates it dynamically from QEMU's internal program representation, translating each basic block as it is encountered during execution; angr [37] transforms machine code to VEX, the IR of the Valgrind framework [29]. In general, IR generation can require a significant amount of work [10], especially when it starts from machine code [21]. Once the IR of the target program is available, a symbolic interpreter can run it and produce symbolic expressions corresponding to each computation. The expressions are typically passed to the symbolic backend for further processing as discussed above; Figure 2 illustrates the process.

2.3 IR-less symbolic execution

While translating target programs to an intermediate representation simplifies the implementation of symbolic execution, *interpreting* IR is much slower than native execution of the corresponding binary, especially in the absence of symbolic data (i.e., when no symbolic reasoning is necessary). This observation has led to the development of Triton [34] and QSYM [45], which follow a different approach: instead of translating the program under test to IR and then interpreting it, they execute the unmodified machine code and instrument it at run time. Concretely, Triton and QSYM both control the target program's execution with Intel Pin [28], a framework for binary instrumentation. Pin provides facilities for inserting custom code when certain machine-code instructions are executed. The symbolic executors use this mechanism to inject code that handles computations symbolically in addition to the concrete computations performed by the CPU. For example, when the CPU is about to add the values contained in

two registers, Pin calls out to the symbolic executor, which obtains the symbolic expressions corresponding to the registers' values, produces the expression that describes the sum, and associates it with the register that receives the result of the computation. See Figure 3 for an overview.

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

However, building symbolic execution directly on machine code has considerable downsides. Most notably, the implementation needs to handle a much larger instruction set: while the IRs that are commonly used for symbolic execution comprise a few dozen different instructions, CPU instruction sets can easily reach hundreds to thousands of them. The symbolic executor has to know how to express the semantics of each of those instructions symbolically, which results in a much more complex implementation. Another problem is architecture dependence: naturally, instrumentation of machine code is a machine-dependent endeavor. IRs, on the other hand, are usually architecture agnostic. IR-based systems therefore work on any architecture where there is a translator from the respective machine code to IR. This is especially relevant for the domain of embedded devices, where a great variety of CPU architectures is in common use. SYMCC uses IR and thus retains the flexibility and implementation simplicity associated with IR-based approaches, yet our compilation-based technique allows it to reach (and surpass) the high performance of IR-less systems, as we show in Section 5.

2.4 Reducing overhead

In either type of symbolic execution, IR-based and IR-less, building symbolic expressions and passing them to the symbolic backend is necessary only when computations involve symbolic data. Otherwise, the result is completely independent of user input and is thus irrelevant for whatever reasoning is performed in the backend. A common optimization strategy is therefore to restrict symbolic handling to computations on symbolic data and resort to a faster execution mechanism otherwise, a strategy that we call *concreteness checks*. In IR-based implementations, symbolic interpretation of IR may even alternate with native execution of machine code on the real or a fast emulated CPU; angr [37], for example, follows this approach. Implementations vary in the scope of their concreteness checks—while QSYM [45] decides whether to invoke the symbolic backend on a per-instruction basis, angr [37] places hooks on relevant operations such as memory and register accesses. Falling back to a fast execution scheme as often as possible is an important optimization, which we also implement in SYMCC (see Section 3.4).

3 Compilation-based symbolic execution

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

3.1 Overview

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

In the context of symbolic execution, current approaches either interpret (in the case of IR-based implementations) or run directly on the CPU but with an attached observer (in IR-less implementations), performing intermittent computations that are not part of the target program. Informally speaking, IR-based approaches are easy to implement and maintain but rather slow, while IR-less techniques reach a high performance but are complex to implement. The core claim of this paper is that we can combine the advantages of both worlds, i.e., build a system that is easy to implement yet fast. To do so, we compile the logic of the symbolic interpreter (or observer) into the target program. Contrary to early implementations of symbolic execution [6, 16, 35], we do not perform this embedding at the source-code level but instead work with the compiler's intermediate representation, which allows us to remain independent of the source language that the program under test is written in, as well as independent of the target architecture (cf. Section 7).

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

Listing 1: An example function in LLVM bitcode. It takes two integers and checks whether the first is exactly twice the second.

To get an intuition for the process, consider the example function in Listing 1. It takes two integers and returns 1 if the first integer equals the double of the second, and 0 otherwise. How would we expect compiler-based symbolic execution to transform the program in order to capture this computation symbolically? Listing 2 shows a possible result. The inserted code calls out to the run-time support library, loaded in the same process, which creates symbolic expressions and eventually passes them to the symbolic backend in order to generate new program inputs (not shown in the example). Note that the transformation inserting those calls happens at compile time; at run time, the program "knows" how to inform the symbolic backend about its computations without requiring any external help and thus without incurring a significant slowdown. Figure 4 summarizes the approach; note how it contrasts with the conventional techniques depicted in Figures 2 and 3. We will now go over the details of the technique.

```
define i32 @is_double(i32, i32) {
  ; symbolic computation
  %3 = call i8* @_sym_get_parameter_expression(i8 0)
  %4 = call i8* @_sym_get_parameter_expression(i8 1)
  %5 = call i8* @_sym_build_integer(i64 1)
  %6 = call i8* @_sym_build_shift_left(i8* %4, i8* %5)
  %7 = call i8* @_sym_build_equal(i8* %6, i8* %3)
  %8 = call i8* @_sym_build_bool_to_bits(i8* %7)

  ; concrete computation (as before)
  %9 = shl nsw i32 %1, 1
  %10 = icmp eq i32 %9, %0
  %11 = zext i1 %10 to i32

  call void @_sym_set_return_expression(i8* %8)
  ret i32 %11
}
```

Listing 2: Simplified instrumentation of Listing 1. The called functions are part of the support library. The actual instrumentation is slightly more complex because it accounts for the possibility of non-symbolic function parameters, in which case the symbolic computation can be skipped.

3.2 Support library

Since we compile symbolic execution capabilities into the target program, all components of a typical symbolic execution engine need to be available. We therefore bundle the symbolic backend into a library that is used by the target program. The library exposes entry points into the symbolic backend

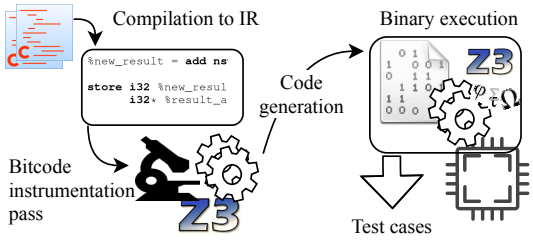

Figure 4: Our compilation-based approach compiles symbolic execution capabilities directly into the target program.

to be called from the instrumented target, e.g., functions to build symbolic expressions and to inform the backend about conditional jumps.

3.3 Symbolic handlers

The core of our compile-time transformation is the insertion of calls to handle symbolic computations. The compiler walks over the entire program and inserts calls to the symbolic backend for each computation. For example, where the target program checks the contents of two variables for equality, the compiler inserts code to obtain symbolic expressions for both operands, to build the resulting "equals" expression and to associate it with the variable receiving the result (see expression `%7` in Listing 2). The code is generated at compile time and embedded into the binary. This process replaces a lot of the symbolic handling that conventional symbolic execution engines have to perform at run time. Our compiler instruments the target program exactly once—afterwards, the resulting binary can run on different inputs without the need to repeat the instrumentation process, which is particularly effective when combined with a fuzzer. Moreover, the inserted handling becomes an integral part of the target program, so it is subject to the usual CPU optimizations like caching and branch prediction.

3.4 Concreteness checks

It is important to realize that each inserted call to the run-time support library introduces overhead: it ultimately invokes the symbolic backend and may put load on the SMT solver. However, involving the symbolic backend is only necessary when a computation receives symbolic inputs. There is no need to inform the backend of fully concrete computations—we would only incur unnecessary overhead (as discussed in Section 2.4). There are two stages in our compilation-based approach where data can be identified as concrete:

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

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

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

4 Implementation of SymCC

We now describe SYMCC, our implementation of compiler-based symbolic execution. We built SYMCC on top of the LLVM compiler framework [25]. Compile-time instrumentation is achieved by means of a custom compiler pass, written from scratch. It walks the LLVM bitcode produced by the compiler frontend and inserts the code for symbolic handling (as discussed in Section 3.3). The inserted code calls the functions exported by the symbolic backend: we provide a thin wrapper around the Z3 SMT solver [11], as well as optional integration with the more sophisticated backend of QSYM [45]. The compiler pass consists of roughly 1,000 lines of C++ code; the run-time support library, also written in C++, comprises another 1,000 lines (excluding Z3 and the optional QSYM code). The relatively small code base shows that the approach is conceptually simple, thus decreasing the probability of implementation bugs.

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

4.1 Compile-time instrumentation

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

Since our compiler extension is implemented as an LLVM pass, it runs in the "middle-end" of LLVM-based compilers—after the frontend has translated the source language into LLVM bitcode but before the backend transforms the bitcode into machine code. SYMCC thus needs to support the instructions and intrinsic functions of the LLVM bitcode language. We implement the same semantics as IR-based symbolic interpreters of LLVM bitcode, such as KLEE [5] and S2E [9]. In contrast to the interpreters, however, we do not *perform* the symbolic computations corresponding to the bitcode instructions at instrumentation time but instead generate code ahead of time that performs them during execution.[2] This means that the instrumentation step happens only once, followed by an arbitrary number of executions. Furthermore, the code that we inject is subject to compiler optimizations and eventually runs as part of the target program, without the need to switch back and forth between the target and an interpreter or attached observer. It is for this reason that we implemented the instrumentation logic from scratch instead of reusing code from KLEE or others: those systems perform run-time instrumentation whereas our implementation needs to instrument the target at compile time.

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

In a recent study, we found that symbolic execution is fastest when it executes at the level of machine code, but that SMT queries are easiest when generated based on the higher-level semantics of an intermediate representation [32]. This is exactly the setup of SYMCC: we reason about computations at the level of LLVM bitcode, but the injected code is compiled down to efficient machine code.

It is sometimes argued that binary-based vulnerability

[2] This also distinguishes our approach from what the formal verification community calls *symbolic compilation* [42]. Symbolic compilers translate the entire program to a symbolic representation in order to reason about all execution paths at once, while we—like all symbolic execution systems—defer reasoning to run time, where it is necessarily restricted to a subset of all possible execution paths.

search is more effective than source-based techniques because it examines the instructions that the processor executes instead of a higher-level representation; it can discover bugs that are introduced during compilation. A full evaluation of this claim is outside the scope of this paper. However, we remark that SYMCC could address concerns about compiler-introduced bugs by performing its instrumentation at the very end of the optimization pipeline, just before code generation. At this point, all compiler optimizations that may introduce vulnerabilities have been performed, so SYMCC would instrument an almost final version of the program—only the code-generation step needs to be trusted. We have not seen the need for such a change in practice, so we leave it to future work.

The reader may wonder whether SYMCC is compatible with compiler-based sanitizers, such as address sanitizer [36] or memory sanitizer [38]. In principle, there is no problem in combining them. Recent work by Österlund et al. shows that sanitizer instrumentation can help to guide fuzzers [31]. We think that there is potential in the analogous application of the idea to symbolic execution—sanitizer checks could inform symbolic execution systems where generating new inputs is most promising. However, our current implementation, like most concolic execution systems, separates test case generation from input evaluation: sanitizers check whether the *current* input leads to unwanted behavior, while SYMCC generates *new* inputs from the current one. We leave the exploration of sanitizer-guided symbolic execution in the spirit of Österlund et al. to future work.

4.2 Shadow memory

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

It would be possible to replace our allocation-tracking scheme with an approach where shadow memory is at a fixed offset from the memory it corresponds to. This is the technique used by popular LLVM sanitizers [36, 38]. It would allow constant-time lookup of symbolic expressions, where currently the lookup time is logarithmic in the number of memory pages containing symbolic data. However, since this number is usually very small (in our experience, below 10), we opted for the simpler implementation of on-demand allocation.

4.3 Symbolic backend

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

However, since the backend is mostly independent from the execution component and only communicates with it via a simple interface, we can replace it without affecting the execution component, our main contribution. We demonstrate this flexibility by integrating the QSYM backend, which can optionally be used instead of our simple Z3 wrapper: We compile a shared library from the portion of QSYM that handles symbolic expressions, link it to our target program and translate calls from the instrumented program into calls to the QSYM code. The interface of our wrapper around the QSYM code consists of a set of functions for expression creation (e.g., `SymExpr _sym_build_add(SymExpr a, SymExpr b)`), as well as helper functions to communicate call context and path constraints; adding a path constraint triggers the generation of new inputs via Z3. Effectively, this means that we can combine all the sophisticated expression handling from the QSYM backend, including dependency tracking between expressions and back-off strategies for hot code paths [45], with our own fast execution component.

4.4 Concreteness checks

In Section 3.4, we highlighted the importance of concreteness checks: for good performance, we need to restrict symbolic reasoning (i.e., the involvement of the symbolic backend) to cases where it is necessary. In other words, when all operands of a computation are concrete, we should avoid any call to the symbolic backend. In our implementation, symbolic expressions are represented as pointers at run time, and the expressions for concrete values are null pointers. Therefore, checking the concreteness of a given expression during execution is a simple null-pointer check. Before each computation in the bitcode, we insert a conditional jump that skips symbolic handling altogether if all operands are concrete; if at least one operand is symbolic, we create the symbolic expressions for the other operands as needed and call out to the symbolic backend. Obviously, when the compiler can infer that a value is a compile-time constant and thus never symbolic at run time, we just omit the generation of code for symbolic handling.

By accelerating concrete computations during symbolic execution, we alleviate a common shortcoming of conventional implementations. Typically, only a few computations in a target program are symbolic, whereas the vast majority of operations involve only concrete values. When symbolic execution introduces a lot of overhead even for concrete com-

putations (as is the case with current implementations despite their concreteness checks), the overall program execution is slowed down considerably. Our approach, in contrast, allows us to perform concrete computations at almost the same speed as in uninstrumented programs, significantly speeding up the analysis. Section 5 shows measurements to support this claim.

4.5 Interacting with the environment

Most programs interact with their environment, e.g., by working with files, or communicating with the user or other processes. Any implementation of symbolic execution needs to either define a boundary between the analyzed program and the (concrete) realm of the operating system, or execute even the operating system symbolically (which is possible in S2E [9]). QSYM [45], for example, sets the boundary at the system call interface—any data crossing this boundary is made concrete.

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

4.6 Supporting additional source languages

Since SYMCC uses the compiler to instrument target programs, it is in principle applicable to programs written in any compiled programming language. Our implementation builds on top of the LLVM framework, which makes it particularly easy to add support for programming languages with LLVM-based compilers, such as C++ [40], Rust [41] and Go [15]. We have implemented C++ support in SYMCC, and we use it as an example for describing the generalized process of adding support for a new source language. The procedure consists of two steps, which we discuss in more detail below: loading our LLVM pass into the compiler and compiling the language's run-time library.

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

4.6.1 Loading the pass

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

4.6.2 Compiling the run-time library

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

For C++ support in SYMCC, we chose `libc++` [26], the LLVM project's implementation of the C++ standard library. It has the advantages that it is easy to build and that it does not conflict with `libstdc++`, the GNU implementation of the library installed on most Linux distributions. Compiling it with SYMCC is a matter of setting the `CC` and `CXX` environment variables to point to SYMCC before invoking the regular build scripts.

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

5 Evaluation

In this section we evaluate SYMCC. We first analyze our system's performance on synthetic benchmarks (Section 5.1),

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

allowing for precisely controlled experiments. Then we evaluate our prototype on real-world software (Section 5.2), demonstrating that the advantages we find in the benchmarks translate to benefits in finding bugs in the real world. The raw data for all figures is available at http://www.s3.eurecom.fr/tools/symbolic_execution/symcc.html.

5.1 Benchmarks

For our benchmarks we use the setup that we proposed in earlier work [32]: at its core, it uses a set of test programs that was published in the course of the DARPA Cyber Grand Challenge (CGC), along with inputs that trigger interesting behavior in each application (called *proofs of vulnerability* or *PoVs*). The same set of programs has been used by Yun et al. in the evaluation of QSYM [45], so we know that QSYM is capable of analyzing them, which enables a fair comparison. We applied the necessary patches for KLEE in order to enable it to analyze the benchmark programs as well.[5] Note that we excluded five programs because they require inter-process communication between multiple components, making them hard to fit into our controlled execution environment, and one more, `NRFIN_00007`, because it contains a bug that makes it behave differently when compiled with different compilers (see Appendix B).

A major advantage of the CGC programs over other possible test sets is that they eliminate unfairness which may otherwise arise from the different instrumentation boundaries in the systems under comparison (see Section 4.5): In contrast with KLEE and QSYM, SYMCC does not currently execute the C standard library symbolically. It would therefore gain an unfair speed advantage in any comparison involving `libc`. The CGC programs, however, use a custom "standard library" which we compile symbolically with SYMCC, thus eliminating the bias.[6]

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

5.1.1 Comparison with other state-of-the-art systems

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

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

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

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

The targets of our comparison are KLEE [5] and QSYM [45]. We decided for KLEE because, like SYMCC, it works on LLVM bitcode generated from source code; an important difference, however, is that KLEE *interprets* the bitcode while SYMCC *compiles* the bitcode together with code for symbolic processing. Comparing with KLEE therefore allows us to assess the value of compilation in the context of symbolic execution. The decision for QSYM is largely motivated by its fast execution component. Its authors demonstrated considerable benefits over other implementations, and our own work provides additional evidence for the notion that QSYM's execution component achieves high performance in comparison with several state-of-the-art systems [32]. Moreover, our reuse of QSYM's symbolic backend in SYMCC allows for a fair comparison of the two systems' execution components (i.e., their frontends). QSYM's approach to symbolic execution requires a relatively complex implementation because the system must handle the entire x86 instruction set—we demonstrate that SYMCC achieves comparable or better performance with a much simpler implementation (and the additional benefit of architecture independence, at the cost of requiring source code or at least LLVM bitcode).

In order to save on the already significant use of computational resources required for our evaluation, we explicitly excluded two other well-known symbolic execution systems: S2E [9] and Driller [39]. S2E, being based on KLEE, is very similar to KLEE in the aspects that matter for our evaluation, and preliminary experiments did not yield interesting insights. Driller is based on angr [37], whose symbolic execution component is implemented in Python. While this gives it distinct advantages for scripting and interactive use, it also makes execution relatively slow [32, 45]. We therefore did not consider it an interesting target for a performance evaluation of symbolic execution.

Pure execution time We executed KLEE, QSYM and SYMCC on the CGC programs, providing the PoVs as input. For the measurement of pure execution time, we did not mark any data as symbolic, therefore observing purely concrete execution inside the symbolic execution engines. In many real-world scenarios, only a fraction of the data in the tested program is symbolic, so efficient handling of non-symbolic (i.e., concrete) computations is a requirement for fast symbolic execution [45]. Figure 5 shows the results: SYMCC executes most programs in under one second (and is therefore almost as

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

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

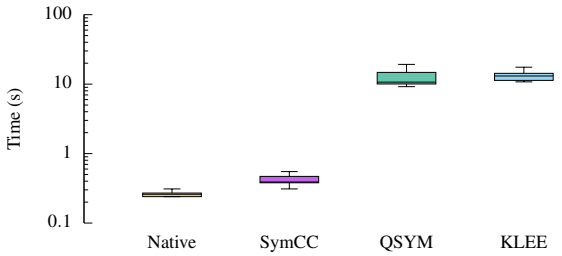

Figure 5: Time spent on pure execution of the benchmark programs, i.e., without symbolic data. Note the logarithmic scale of the time axis. "Native" is the regular execution time of the uninstrumented programs. On average, SYMCC is faster than QSYM by 28× and faster than KLEE by 30× (KLEE can execute only 56 out of 116 programs).

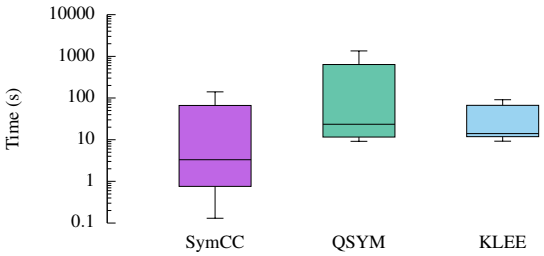

Figure 6: Time spent on concolic execution of the benchmark programs, i.e., with symbolic inputs (logarithmic scale). SYMCC is faster than QSYM by an average factor of 10× and faster than KLEE by 12× (KLEE can execute only 56 out of 116 programs).

fast as native execution of uninstrumented programs), while QSYM and KLEE need seconds up to minutes.

Execution time with symbolic inputs Next, we performed concolic execution on the CGC programs, again using the PoVs as input. This time, we marked the input data as symbolic, so that symbolic execution would generate new test cases along the path dictated by each PoV. For a fair comparison, we configured KLEE to perform concolic execution like QSYM and SYMCC. This setup avoids bias from KLEE's forking and scheduling components [32]. It is worth noting, however, that KLEE still performs some additional work compared to QSYM and SYMCC: since it does not rely on external sanitizers to detect bugs, it implements similar checks itself, thus putting more load on the SMT solver. Also, it features a more comprehensive symbolic memory model. Since these are intrinsic aspects of KLEE's design, we cannot easily disable them in our comparison.

In essence, all three symbolic execution systems executed the target program with the PoV input, at each conditional attempting to generate inputs that would drive execution down the alternative path. The results are shown in Figure 6: SYMCC is considerably faster than QSYM and KLEE even in the presence of symbolic data.

Coverage Finally, we measured the coverage of the test cases generated in the previous experiment using the methodology of Yun et al. [45]: for each symbolic execution system, we recorded the combined coverage of all test cases per target program in an AFL coverage map [46].[7] On each given target program, the result was a set of covered program points for each system, which we will call S for SYMCC and R for the system we compare to (i.e., KLEE or QSYM). We then assigned a score d in the range $[-1.0, 1.0]$ as per Yun et al. [45]:

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

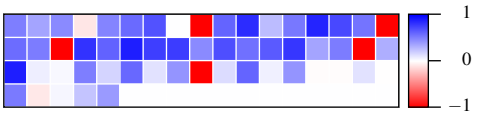

Figure 7: Coverage score comparing SYMCC and KLEE per tested program (visualization inspired by Yun et al. [45]): blue colors mean that SYMCC found more paths, red colors indicate that KLEE found more, and white symbolizes equal coverage. SYMCC performs better on 46 programs and worse on 10 (comparison restricted to the programs that KLEE can execute, i.e., 56 out of 116).

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

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

We visualize the coverage score per test program in Figures 7 and 8. The former shows that SYMCC generally achieves a higher coverage level than KLEE; we mainly

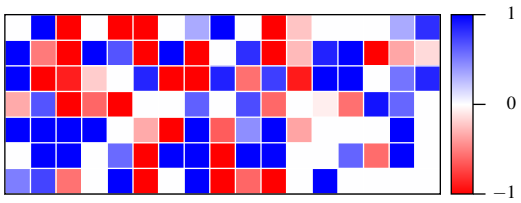

Figure 8: Comparison of coverage scores between SYMCC and QSYM. SYMCC found more paths on 47 programs and less on 40; they discovered the same paths on 29 programs. Similar coverage is expected because SYMCC uses the same symbolic backend as QSYM.

attribute differences to the significantly different symbolic backends. The latter demonstrates that SYMCC's coverage is comparable to QSYM's, i.e., the compilation-based execution component provides information of comparable quality to the symbolic backend. We suspect the reason that coverage of some programs differs at all—despite the identical symbolic backends in QSYM and SYMCC—is twofold:

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

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

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

5.1.2 Initialization overhead

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

$$T_{\text{SYMCC}} = I_{\text{SYMCC}} + A_{\text{SYMCC}} \tag{1}$$

$$T_{\text{other}} = I_{\text{other}} + A_{\text{other}} = I_{\text{other}} + S \cdot A_{\text{SYMCC}} \tag{2}$$

Consequently, we can compute the speedup factor as follows:

$$S = \frac{T_{\text{other}} - I_{\text{other}}}{T_{\text{SYMCC}} - I_{\text{SYMCC}}} \tag{3}$$

In order to obtain accurate predictions for the analysis time of *long-running* programs, we therefore need to take the initialization time into account when computing the speedup factor. As a simple approximation for the worst case from SYMCC's point of view, we assumed that the shortest observed execution consists of initialization only, i.e., suppose A_{SYMCC} and A_{other} are zero in the analysis of the fastest-running program. In other words, for each system we subtracted the time of the fastest analysis observed in Section 5.1.1 from all measurements. Then we recomputed the speedup in the affine model presented above. For concolic execution with KLEE, we obtained an average factor of 2.4 at a constant-time overhead of 9.20 s, while for QSYM we computed a factor of 2.7 at a constant-time overhead of 9.15 s. SYMCC's constant-time overhead is 0.13 s; this confirms the benefit of instrumenting the target at compile time.

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

5.1.3 Compilation time and binary size

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

Next, we compared the size of each instrumented executable produced by SYMCC with the corresponding unmodified executable emitted by `clang`. On average, our instrumented binaries are larger by a factor of 3.4. While we have not optimized SYMCC for binary size, we believe that there

[8] The documentation of AFL, for example, recommends that target programs should be fast enough to achieve "ideally over 500 execs/sec most of the time" [46].

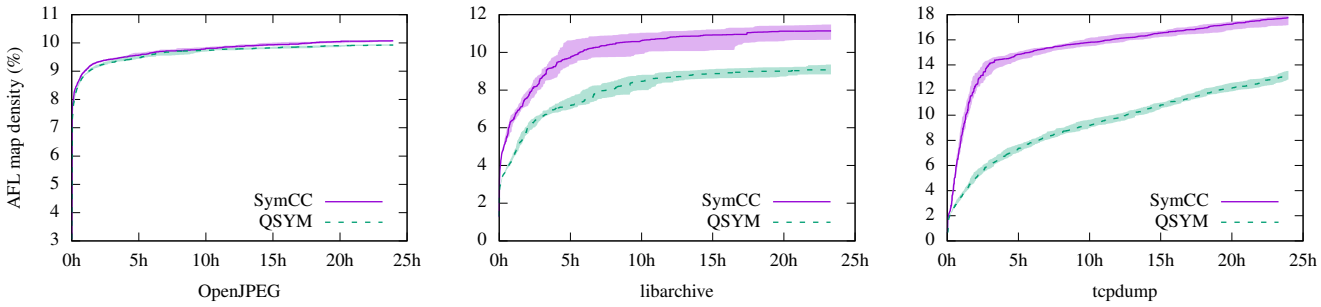

Figure 9: Density of the AFL coverage map over time. The shaded areas are the 95 % confidence corridors. The respective differences between QSYM and SYMCC are statistically significant with $p < 0.0002$. Note that the coverage improvement correlates with the speedup displayed in Figure 10.

is potential to reduce this factor if needed. The largest contribution to code size comes from run-time concreteness checks; if binary size became a major concern, one could disable concreteness checks to trade execution time for space. In our tests we have not experienced the necessity.

5.1.4 Impact of concreteness checks

In Section 3.4, we claimed that considerable improvements can be gained by checking data for concreteness at run time and skipping symbolic computation if all operands are concrete.

To illustrate this claim, let us examine just the initialization phase of the CGC program `CROMU_00001`. During the startup, the CGC "standard library" populates a region in memory with pseudo-random data obtained by repeated AES computations on a seed value; this happens before any user input is read. In the uninstrumented version of the program, the initialization code executes within roughly 8 ms. This is the baseline that we should aim for. However, when we run a version of SYMCC *with concreteness checks disabled* on `CROMU_00001`, execution takes more than five minutes using our own simple backend, and with the faster QSYM backend SYMCC still requires 27 s. The reason is that the instrumented program calls into the symbolic backend at every operation, which creates symbolic expressions, regardless of the fact that all operands are fully concrete. The QSYM backend performs better than our simple backend because it can fold constants in symbolic expressions and has a back-off mechanism that shields the solver against overload [45]. However, recall that we are executing on concrete data only—it should not be necessary to invoke the backend at all!

In fact, concreteness checks can drastically speed up the analysis by entirely freeing the symbolic backend from the need to keep track of concrete computations. With concreteness checks enabled (as described in Section 4.4), the symbolic backend is only invoked when necessary, i.e., when at least one input to a computation is symbolic. For the initialization of `CROMU_00001`, enabling concreteness checks results in a reduction of the execution time to 0.14 s with the QSYM backend (down from 27 s). The remaining difference with the uninstrumented version is largely due to the overhead of backend initialization and memory operations for book-keeping.

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

5.2 Real-world software

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

We used QSYM and SYMCC in combination with the fuzzer AFL [46] to test popular open-source projects (using AFL version 2.56b); KLEE is not applicable because of unsupported instructions in the target programs. For each target program, we ran an AFL master instance, a secondary AFL instance, and one instance of either QSYM or SYMCC. The symbolic execution engines performed concolic execution on the test cases generated by the AFL instances, and the resulting new test cases were fed back to the fuzzers. Note that this is a relatively naive integration between symbolic execution and fuzzer; however, since the focus of this work is on the performance of symbolic execution, we leave the exploration of more sophisticated coordination techniques to future work.

Fuzzing is an inherently randomized process that introduces a lot of variables outside our control. Following the recommendations by Klees et al. [23], we therefore let the analysis continue for 24 hours, we repeated each experiment 30 times, and we evaluated the statistical significance of the results using the Mann-Whitney U test. Our targets are OpenJPEG, which we tested in an old version with known vul-

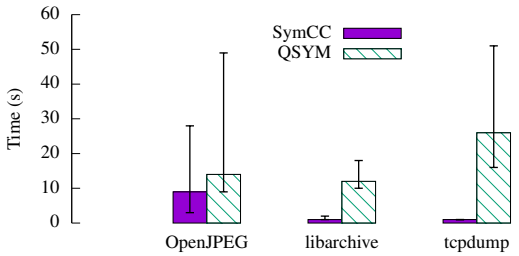

Figure 10: Time per symbolic execution (median and quartiles, excluding executions that exhausted time or memory resources). The difference between QSYM and SYMCC is statistically significant with $p < 0.0001$. Note the correlation between higher speed here and increased coverage in Figure 9.

nerabilities, and the latest master versions of libarchive and tcpdump. In total, we spent 3 experiments $\times$ 2 analysis systems $\times$ 30 $\frac{\text{iterations}}{\text{experiment·analysis system}} \times 3\frac{\text{CPU cores}}{\text{iteration}} \times$ 24 hours = 12 960 CPU core hours $\approx$ 17.8 CPU core months. The hardware used for these experiments was an Intel Xeon Platinum 8260 CPU with 2 GB of RAM available to each process (AFL, QSYM or SYMCC).

While running the fuzzer and symbolic execution as specified above, we measured the code coverage as seen by AFL[9] (Figure 9) and the time spent on each symbolic execution of the target program (Table 1 and Figure 10). We found that SYMCC not only executes faster than QSYM (which is consistent with the benchmarks of Section 5.1) but also reaches significantly higher coverage on all three test programs. Interestingly, the gain in coverage appears to be correlated with the speed improvement, which confirms our intuition that accelerating symbolic execution leads to better program testing.

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

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

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

[9] AFL's coverage map is known to be prone to collisions and therefore does not reflect actual code coverage [14]. However, AFL bases its decisions on the coverage map, so the map is what counts when evaluating the benefit of a symbolic execution system for the fuzzer.

6 Discussion and future work

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

6.1 Benefits of compilation

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

6.2 Portability and language support

Our current prototype supports programs written in C and C++. However, since we build on the LLVM framework, we could support any program that is translatable to LLVM bitcode. In particular, this means that we can integrate SYMCC into any LLVM-based compiler, such as the default compilers for Rust [41] and Swift [1], and the alternative Go compiler `gollvm` [15]. Similarly, we can generate binaries for any machine architecture that LLVM supports, without any changes in our code. More generally, the technique of compilation-based symbolic execution applies to any compiled programming language.

6.3 Binary analysis

So far, we have only discussed compilation-based symbolic execution in contexts where the source code of the program

	OpenJPEG		libarchive		tcpdump	
	SYMCC	QSYM	SYMCC	QSYM	SYMCC	QSYM
Average execution time per analysis (s)	1.9	14.9	1.6	19.1	0.3	27.1
Average solver time per analysis (s)	26.4	15.7	0.2	1.8	0.3	8.2
Average total time per analysis (s)	28.3	30.6	1.8	20.9	0.6	35.3
Average share of execution (%)	6.7	48.7	91.7	91.2	41.7	76.8
Average share of SMT solving (%)	93.3	51.3	8.3	8.8	58.3	23.2
Speedup factor vs QSYM	1.1		11.6		58.8	

Table 1: Time split between execution and SMT solving. See Figure 10 for a visualization of the total analysis times. Note how the speedup factor in the last row correlates with SYMCC's improved coverage displayed in Figure 9.

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

6.3.1 Lifting

SYMCC currently uses compiler frontends to create LLVM bitcode from source code, but there is no fundamental reason for creating the bitcode from the source: S2E [9] popularized the idea of generating a high-level IR from binaries for the purpose of symbolic execution. It generates LLVM bitcode from the internal program representation of QEMU [2] and runs it in KLEE [5]. A similar approach is used by angr [37], which dynamically generates VEX IR for a symbolic interpreter from binaries. Several other such *lifters* have been designed for purposes outside the realm of symbolic analysis [21]. While the IR obtained from binaries is more verbose [32], SYMCC could be used in combination with a lifter to compile symbolic handling into existing binaries. Trail of Bits has recently applied a similar lifting technique to KLEE, essentially converting it from a source-based tool to a symbolic execution engine that can work on binaries [43].

6.3.2 Hybrid with QSYM

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

We would like to remark, however, that even binary-based symbolic execution is often evaluated on open-source software, and many gray-box fuzzers like AFL [46] only reach their full performance when the source code of the program under test is available for instrumentation.

7 Related work

As a program analysis technique, symbolic execution exists on a spectrum. On the one extreme of that spectrum, bounded model checking inlines all functions, unrolls loops up to a certain bound and translates the entire program into a set of constraints [13, 33]. While this process is sometimes called "symbolic compilation" [3], it is not to be confused with our compilation-based symbolic execution: bounded verification reasons about all executions at once, thus allowing for very sophisticated queries but pushing most of the load to the SMT solver. Our approach, in contrast, follows the tradition of symbolic execution by reasoning about the program per execution path [5, 9, 37]. On the other end of the spectrum, fuzz testing executes target programs with very light or no instrumentation, heuristically mutating inputs (and possibly using feedback from the instrumentation) in the hope of finding inputs that evoke a certain behavior, typically program crashes [4, 8, 12, 27, 46].

While bounded verification provides powerful reasoning capabilities, fuzzing is extremely fast in comparison. Conventional symbolic execution lies between the two [5, 9, 37], with state-merging approaches [24, 42] leaning more towards bounded verification, and hybrids with fuzzing attempting to produce fast but powerful practical systems [39, 45]. It is this last category of systems that forms the basis for our approach: we aim at a combination of symbolic execution and fuzzing similar to Driller [39] and QSYM [45]. By speeding up symbolic execution, we aim to make its more sophisticated reasoning available in situations where previously only fuzzing was fast enough.

Current work in symbolic execution, as outlined above and referenced throughout the paper, applies either interpreter- or observer-based techniques. While early systems embedded symbolic reasoning directly [6, 16, 35], they performed the instrumentation at the level of C code, which severely restricts the set of possible input programs and complicates the implementation significantly [16]. The approach of instrumenting the program under test directly was abandoned in KLEE [5], and subsequent work in symbolic execution mostly followed its lead. We are not aware of any performance comparison between the direct embedding implemented in early work and the interpreter approach to symbolic execution implemented by KLEE and later systems; we assume that the switch happened because interpreters are more flexible and easier to implement correctly. With SYMCC, we demonstrate that directly embedding concolic execution into the target program yields much higher performance than state-of-the-art systems; at the same time, however, performing the embedding at the level of the compiler's intermediate representation allows us to maintain the flexibility that is common in modern implementations.

The most closely related project outside the field of symbolic execution is Rosette, a "solver-aided programming language" [42]. It allows programmers to express symbolic constraints while writing a program, which it then executes in a "Symbolic Virtual Machine". In contrast to our approach, it is not meant for the analysis of arbitrary programs but rather aims to support the development of program-synthesis and verification tools. It requires the developer to use a domain-specific language and design the program for symbolic analysis from the start. Moreover, it does not compile the program to machine code but rather executes it in a host environment, similarly to how KLEE orchestrates multiple execution states in a single process.

SMT Kit [19] is a project that performs a similar embedding into C++, and there is (incomplete) support for automatically transforming source code to use the library [18]. The idea, if fully executed, may have led to a system similar to SYMCC, but the project seems to have been abandoned years ago without a publication, and we have been unable to contact the author. We anticipate that a robust source-to-source translation would have been much more difficult to implement than our IR transformation due to the complexity of the C++ language in comparison with LLVM bitcode. Moreover, the system would have been inherently limited to a single programming language, just like the early implementations for C mentioned above, while SYMCC's transformation at the IR level allows it to support any source language for which an LLVM-based compiler exists.

8 Conclusion

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

Acknowledgments

We would like to thank Insu Yun, the first author of QSYM, for helping us to replicate the experimental results reported in the QSYM paper [45]. Moreover, we are grateful to Khaled Yakdan for his feedback on earlier versions of this paper. Finally, we thank the anonymous paper and artifact reviewers for taking the time to study our work and provide constructive feedback. This work has been supported by the DAPCODS/IOTics ANR 2016 project (ANR-16-CE25-0015).

Availability

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

References

[1] Apple Inc. Swift.org – compiler and standard library. https://swift.org/compiler-stdlib/#compiler-architecture.

[2] Fabrice Bellard. QEMU, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX Track*, volume 41, page 46, 2005.

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

[4] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on*

Computer and Communications Security, pages 2329–2344. ACM, 2017.

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

[6] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. EXE: automatically generating inputs of death. *ACM Transactions on Information and System Security (TISSEC)*, 12(2):10, 2008.

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

[8] Hongxu Chen, Yinxing Xue, Yuekang Li, Bihuan Chen, Xiaofei Xie, Xiuheng Wu, and Yang Liu. Hawkeye: Towards a desired directed grey-box fuzzer. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 2095–2108. ACM, 2018.

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

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

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

[12] Joe W. Duran and Simeon Ntafos. A report on random testing. In *Proceedings of the 5th International Conference on Software Engineering*, ICSE '81, pages 179–183, Piscataway, NJ, USA, 1981. IEEE Press.

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

[14] Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuwen Tu, Kang Li, Zhongyu Pei, and Zuoning Chen. CollAFL: Path sensitive fuzzing. In *2018 IEEE Symposium on Security and Privacy*, pages 679–696. IEEE, 2018.

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

[16] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: directed automated random testing. In *ACM Sigplan Notices*, volume 40, pages 213–223. ACM, 2005.

[17] Patrice Godefroid, Michael Y. Levin, and David Molnar. Sage: whitebox fuzzing for security testing. *Communications of the ACM*, 55(3):40–44, 2012.

[18] Alex Horn. Clang CRV front-end. https://github.com/ahorn/native-symbolic-execution-clang, 2014.

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

[20] C.-A. Hsieh, M. T. Conte, T. L. Johnson, J. C. Gyllenhaal, and W.-W. Hwu. Compilers for improved java performance. *Computer*, 30(6):67–75, June 1997.

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

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

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

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

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

[26] LLVM Project. "libc++" C++ standard library. https://libcxx.llvm.org/.

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

[28] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building

customized program analysis tools with dynamic instrumentation. In *Acm sigplan notices*, volume 40, pages 190–200. ACM, 2005.

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

[30] Anh Nguyen-Tuong, David Melski, Jack W. Davidson, Michele Co, William Hawkins, Jason D. Hiser, Derek Morris, Ducson Nguyen, and Eric Rizzi. Xandra: An autonomous cyber battle system for the cyber grand challenge. *IEEE Security & Privacy*, 16(2):42–51, 2018.

[31] Sebastian Österlund, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Parmesan: Sanitizer-guided greybox fuzzing. In *29th USENIX Security Symposium (USENIX Security 20)*, 2020.

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

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

[34] Florent Saudel and Jonathan Salwan. Triton: A dynamic symbolic execution framework. In *Symposium sur la sécurité des technologies de l'information et des communications, SSTIC, France, Rennes, June 3-5 2015*, pages 31–54. SSTIC, 2015.

[35] Koushik Sen, Darko Marinov, and Gul Agha. CUTE: a concolic unit testing engine for c. In *ACM SIGSOFT Software Engineering Notes*, volume 30, pages 263–272. ACM, 2005.

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

[37] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, et al. SoK: (State of) The art of war: Offensive techniques in binary analysis. In *2016 IEEE Symposium on Security and Privacy*, pages 138–157. IEEE, 2016.

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

[39] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *NDSS*, volume 16, pages 1–16, 2016.

[40] The Clang Team. Clang C language family frontend for LLVM. https://clang.llvm.org/, 2019.

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

[42] Emina Torlak and Rastislav Bodik. A lightweight symbolic virtual machine for solver-aided host languages. In *ACM SIGPLAN Notices*, volume 49, pages 530–541. ACM, 2014.

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

[44] Clark Wiedmann. A performance comparison between an apl interpreter and compiler. *SIGAPL APL Quote Quad*, 13(3):211–217, March 1983.

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

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

A SymCC usage example

Figure 4 shows an example interaction with SymCC: We first compile the program displayed in Listing 3, simulating a log-in interface. Then we run the program with an initial test input and demonstrate that concolic execution generates a new test input that allows us to access the most interesting portion of the program. While this is a very basic example, we hope that it gives the reader an idea of how SymCC can be used.

```
#include <iostream>

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

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

  return 0;
}
```

Listing 3: A sample C++ program that emulates a log-in interface. The most interesting portion of the program is reached when the user inputs "root".

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

Listing 4: A shell session that demonstrates how a user would compile and run the program from Listing 3 with SYMCC. Lines prefixed with a dollar sign indicate commands entered by the user. Note how the analysis proposes "root" as a new test input.

In larger software projects, it is typically sufficient to export `CC=symcc` and `CXX=sym++` before invoking the respective build system; it will pick up the compiler settings and build an instrumented target program transparently.

B The curious case of `NRFIN_00007`

The CGC program `NRFIN_00007` contains a bug that changes the program's observable behavior depending on the compiler and compilation flags. We believe that it is unrelated to the intended vulnerability in the program (i.e., a buffer overflow triggered by certain user inputs). Listing 5 shows an excerpt of the program's `main` function. During initialization (and before any user input is read), it checks the uninitialized variable `ret` and exits prematurely if its value is non-zero. In practical execution, this causes the program to exit early depending on the stack layout chosen by the compiler. Since SYMCC, KLEE and QSYM all use different means to compile the target program, the bug would introduce errors into our evaluation; we therefore excluded `NRFIN_00007` from the test set.

```
int main(void) {
    int ret;
    size_t size;

    malloc_init();

    if (ret != 0)
        _terminate(ret);

    // ...
}
```

Listing 5: A bug in the code of `NRFIN_00007`. The variable `ret` is used uninitialized; if its value is non-zero, the program exits prematurely without ever reading user input.