SymFusion: Hybrid Instrumentation for Concolic Execution

Emilio Coppa
coppa@diag.uniroma1.it
Sapienza University of Rome
Italy

Heng Yin
heng.yin@ucr.edu
University of California, Riverside
USA

Camil Demetrescu
demetres@diag.uniroma1.it
Sapienza University of Rome
Italy

ABSTRACT

Concolic execution is a dynamic twist of symbolic execution designed with scalability in mind. Recent concolic executors heavily rely on program instrumentation to achieve such scalability. The instrumentation code can be added at compilation time (e.g., using an LLVM pass), or directly at execution time with the help of a dynamic binary translator. The former approach results in more efficient code but requires recompilation. Unfortunately, recompiling the entire code of a program is not always feasible or practical (e.g., in presence of third-party components). On the contrary, the latter approach does not require recompilation but incurs significantly higher execution time overhead.

In this paper, we investigate a *hybrid* instrumentation approach for concolic execution, called SymFusion. In particular, this hybrid instrumentation approach allows the user to recompile the core components of an application, thus minimizing the analysis overhead on them, while still being able to dynamically instrument the rest of the application components at execution time. Our experimental evaluation shows that our design can achieve a nice balance between efficiency and efficacy on several real-world applications.

KEYWORDS

symbolic execution, code instrumentation

ACM Reference Format:
Emilio Coppa, Heng Yin, and Camil Demetrescu. 2022. SymFusion: Hybrid Instrumentation for Concolic Execution. In *37th IEEE/ACM International Conference on Automated Software Engineering (ASE '22), October 10–14, 2022, Rochester, MI, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3551349.3556928

This paper is dedicated to the memory of Camil Demetrescu, a brilliant researcher and a great teacher that has inspired many students and colleagues. I was lucky enough to be one of his students and then one of his research collaborators for several years. I will carry the memory of Camil with me for the rest of my life.

Emilio

1 INTRODUCTION

Symbolic execution [2, 13, 30, 38] is a popular software testing technique that executes a program over *symbolic*, rather than *concrete*, i.e., fixed a priori, inputs. The technique builds symbolic expressions to represent the computations over symbolic terms and then queries an SMT solver to evaluate which branch directions can be taken by the program when assigning the symbolic inputs. While this approach naturally supports analyses aiming at code coverage [11], it can also be valuable for several security tasks, such as vulnerability detection [14, 36, 44], exploit generation [1], and reverse engineering [6, 41]. Unfortunately, a limiting factor for this technique is its low scalability.

Concolic execution [25, 44] is a dynamic twist of symbolic execution where the program is concretely executed over one input and the analysis is carried out only over the explored path. Any symbolic branch condition met along the path can be then *negated* to generate alternative inputs, which can be later used for other concolic explorations. While numerous ideas (§2) help concolic execution scale on large programs, efficient program instrumentation is crucial to track the symbolic state with minimal overhead.

The instrumentation code used by recent concolic executors is typically added into a program using two strategies: either at *compilation time*, e.g., through an LLVM pass, as in SymCC [36], or at *execution time* using a dynamic binary translator (DBT), as done by SymQEMU [37] and other recent tools [8, 44].

Instrumentation at compilation time typically generates more efficient code as the injected code can be seamlessly mixed with the application code, and then benefit from the powerful optimizations available in modern compilers. However, it requires to recompile the program code with a custom compiler toolchain. Unfortunately, while we may expect developers to be able to recompile the core components of an application, they may struggle to recompile third-party libraries, including the ones provided by the operating system. When a component is not instrumented, the symbolic expressions may be inaccurate. To mitigate such problem, tools may devise function models [11, 42]: each model mimics the effects of the uninstrumented code on the symbolic state. Unfortunately, such models are still mostly written by hand, often leading to inaccurate and incomplete implementations [3].

Instrumentation at execution time instead does not require to recompile the program, since a DBT can dynamically instrument the program during its execution, but it typically generates simpler, less optimized, instrumentation code. Additionally, DBTs introduce a non-negligible overhead when executing a program as they need to closely control its execution, perform JIT translation, and track the program state by maintaining a *virtual* CPU state. These aspects may significantly increase the analysis overhead: for instance, as detailed in §2, SymQEMU could be 6.5× slower than SymCC on a simple code snippet.

Our contributions. In this paper, we investigate whether it is possible to devise a new concolic executor based on a mix, or a *fusion*, of different instrumentation strategies. In other words, we explore a

ASE '22, October 10–14, 2022, Rochester, MI, USA

ACM ISBN 978-1-4503-9475-8/22/10…$15.00
https://doi.org/10.1145/3551349.3556928

hybrid instrumentation approach, where the core components of an application can be instrumented at compilation time (as in SymCC), while the remaining components can be dynamically instrumented at execution time (as in SymQEMU). The goal is to achieve the benefits of both approaches in terms of efficiency and flexibility.

Although the idea may seem simple, we face several challenges. First, the approach has to accurately identify when the program is moving from code that was instrumented at compilation time to code that was not instrumented and thus requires to be executed under the DBT. Second, the two instrumentation strategies look at the program from different perspectives: for instance, an LLVM pass works on an architecture-independent representation that operates on the program's *values*, while a DBT transforms the binary code, which is necessarily tightly coupled with the low-level aspects of the underlying architecture, such as the registers and the platform ABI. Our approach thus needs to build a *bridge* across these two perspectives, supporting a seamless propagation of the symbolic state. Finally, existing tools hide some unexpected gaps that require close attention to make the overall approach effective. For instance, SymQEMU ignores the effects of some platform-specific instructions, possibly losing track of the symbolic expressions.

In more detail, the contributions of the paper are:

- An investigation (§2) of the advantages and disadvantages of the two instrumentation strategies when considering them in the context of concolic execution.
- A new design (§3) for a concolic executor based on hybrid instrumentation, called SymFusion. Our tool uses an LLVM pass to instrument the core components at compilation time and QEMU to instrument the other code at execution time. We also present some low-level optimizations (§4).
- An experimental evaluation (§5) which first validates our design through several microbenchmarks and then analyzes the performance of SymFusion on several complex real-world applications considering different experimental scenarios. In particular, we compare SymFusion with respect to SymCC and SymQEMU in terms of efficiency (i.e., analysis time) and effectiveness (i.e., how valuable are the inputs generated by a tool). We show that SymFusion is indeed faster than SymQEMU and more effective than SymCC.

Release of the prototype. To facilitate extensions of our approach, we make our contributions available at:

https://season-lab.github.io/SymFusion/

2 BACKGROUND AND RELATED WORK

The main ideas behind SymFusion arise from different existing techniques and prior works. We now review some of them.

Program instrumentation. The code of an application can be instrumented in different ways [29]. When the source code is available, the code can be augmented before compilation using language-specific tools [12, 39], or transformed during the compilation, by working on the compiler intermediate representation (IR). Several recent tools [24, 36, 40] have favored this strategy devising passes for LLVM [31], an extremely powerful compiler toolchain.

When only the program binary code is available, instrumentation can be done statically [23], i.e., before executing the program, or dynamically, i.e., at execution time, using a Dynamic Binary Translator (DBT) [34]. Since static binary instrumentation is a less common approach in concolic execution, we do not explicitly discuss it, although, it shares some traits with the dynamic strategy. To make binary instrumentation easier to port across platforms, binary tools may first *lift* the code into a more architecture-independent IR, inject additional IR statements, and then *lower* back the result into platform-dependent code. Unfortunately, IRs from DBTs are still tight to several low-level aspects, requiring to reason, e.g., over specific registers and platform-dependent rules.

QEMU [4] is a well-known machine emulator and virtualizer, which internally integrates a DBT based on the Tiny Code Generation (TCG) IR. When executed in *user mode*, QEMU operates over a single application. To keep track of the program execution state, it generates code that updates a virtual CPU state, which is kept in memory. Hence, when an instrumented basic block is executed, it first reads the virtual registers, manipulates them using native registers, and then writes back their values into the memory.

Concolic execution. Symbolic execution [2, 13] is a very powerful program analysis technique, which evaluates the program behavior on symbolic, rather than concrete, inputs. Any program computation involving the inputs is represented using symbolic expressions. When the program reaches a branch decision, the symbolic executor uses an SMT solver [7, 16, 22] to evaluate which directions (true or false) can be taken when assigning the input values. When both directions are feasible, symbolic execution forks the state and explores the paths in parallel. A symbolic executor can be implemented as an interpreter of the program code [11]: however, this may result in non-negligible overhead [35, 36]. Moreover, a symbolic executor strongly relies on the solver to understand how to continue the exploration, possibly limiting the analysis progress in some cases.

To mitigate these problems, concolic execution [25, 44] devises a different strategy. The program is executed concretely over an input, thus exploring one single path at a time. Along the exploration, the executor builds the symbolic expressions and queries an SMT solver to generate alternative inputs. For each alternative input, a new exploration can be performed, allowing it to explore several paths. A notable benefit of concolic execution is that the concrete state is implicitly maintained by the native CPU, considerably reducing the work for the executor. Additionally, the exploration can go on even when the solver is unable to answer some queries, since the native execution drives the exploration. One downside of concolic execution is that it has to *repeat* work across the executions, possibly incurring even more overhead than symbolic execution. Concolic executors thus require efficient instrumentation code to reduce their overhead. Recent examples of concolic executors exploiting instrumentation at compilation time are SymCC [36] and SymSan [15], while relevant examples of concolic executors using instrumentation at execution time are QSYM [44], Fuzzolic [8], and SymQEMU [37].

Instrumentation for concolic execution. Figure 1 exemplifies how SymCC and SymQEMU may instrument a simple excerpt of code.

SymCC exploits an LLVM pass: it can thus exploit the knowledge available in the compiler. For instance, it may know that the variable b does not depend on the input and thus can be considered concrete.

```
// original code
a = a + b; // compiler knows that b is not input-dependent

// when instrumenting it with SymCC
if (a_expr) // is variable a a symbolic?
  a_expr = _sym_build_add(a_expr, _sym_build_int(b));
a = a + b;

// when instrumenting it with SymQEMU
int a = vcpu->reg1;                       // memory access
int b = vcpu->reg2;                       // memory access
void* a_expr = vcpu->reg1_expr;           // memory access
void* b_expr = vcpu->reg2_expr;           // memory access
a_expr = helper_sym_add(a, a_expr, b, b_expr);
a = a + b;
vcpu->reg1 = a;                           // memory access
vcpu->reg1_expr = a_expr;                 // memory access
```

Figure 1: SymCC vs SymQEMU: instrumentation example.

```
int count = 0, N = 15000, x = input();
for (int i = 0; i < N; i++) count += x;
if (count == 2 * N) reach_me();
```

Figure 2: How fast can a concolic executor analyze this code?

Its instrumentation can then only test whether a is symbolic: this is done by testing if the *shadow* variable of a, called a_expr, is not NULL. When true, it builds a new symbolic expression using two functions from the symbolic runtime. Thanks to the compiler, the variables can be efficiently tracked by native registers.

SymQEMU instruments the code at execution time using QEMU, which tracks the execution state using a virtual CPU state. For each virtual register (e.g., vcpu->reg1), SymQEMU adds a *shadow* virtual register (e.g., vcpu->reg1_expr) to track the symbolic expression associated with the register. The instrumented code hence first needs to load from memory the virtual registers for a and b and the associated shadow registers. Then it tests whether they are symbolic. Since the basic blocks are typically translated independently from each other, a DBT may fail to realize that b cannot be symbolic, requiring the injected code to test it explicitly. Additionally, DBTs may support only branchless instrumentation within a basic block. Hence, conditional tests must be delegated to helper functions: the code thus *always* performs the call to helper_sym_add. Finally, the code must save the virtual registers back into memory.

Figure 2 shows a more involved snippet of C code. Using concolic execution, it is possible to automatically identify that the function reach_me is executed when $x = 2$. SymCC can analyze the code in 0.83 seconds on an AMD Ryzen 5900XT CPU running Ubuntu 20.04. On the same platform, SymQEMU, which does not require to recompile the code with a custom toolchain, takes 5.39 seconds, i.e., 6.5× slower than SymCC. Both tools perform the same number of queries and use the same *symbolic runtime*, i.e., the software component in charge of building and reasoning the expressions. The main difference is due to: (a) the overhead introduced by the QEMU DBT to track and instrument the program execution, and (b) the efficiency of the instrumentation code. In this example, these aspects are exacerbated by the presence of a CPU-intensive loop.

```
int x = input();
int r = lib_identity_fn(x); // fn from an external lib
if (r == 23) reach_me();
```

Figure 3: Input propagation through uninstrumented code.

In concolic execution, failing to instrument some code may lead to inaccurate symbolic expressions. For instance, consider Figure 3, where lib_identity_fn is a function that just returns its argument, i.e., $r \leftarrow x$. The function reach_me is thus executed when the input is equal to 23. If we assume that lib_identity_fn is within an uninstrumented library, then SymCC fails to understand that $r = x$. On the other hand, SymQEMU does not have any issue identifying such dependency but its analysis may be slower.

Comparison of instrumentation strategies. We now summarize more in general the advantages (⊕) and disadvantages (⊖) of different instrumentation strategies in the context of concolic execution.

Instrumentation at compilation time comes with different traits:

- ⊕ No overhead at execution time to instrument the code.
- ⊕ The instrumentation pass works on the compiler IR, which is typically mature and well-defined. Additionally, it is relatively easy to perform even complex code transformations.
- ⊕ The instrumented code is well-optimized and concise. In particular, the pass can exploit code analyses and high-level knowledge available during the compilation. Moreover, the generated code can benefit from the optimization pipeline.
- ⊕ The pass can easily support several platforms.
- ⊖ The code must be recompiled with a custom compiler toolchain.
- ⊖ The placement of the pass within the compilation pipeline is not trivial. Being early in the pipeline makes it possible to benefit from subsequent optimizations but the final program code may be quite different than what is seen by the pass. For instance, the compiler may replace calls to specific functions with specialized inline code: if this code is not instrumented then these transformations are *destructive* for the concolic analysis. Being late in the pipeline may mitigate this problem but the injected code does not benefit from most optimizations.
- ⊖ When a function is not instrumented, then a model must be created for it. Otherwise, the symbolic state could be inaccurate.
- ⊖ The compiler IR can be very rich, integrating several complex and specialized data types (e.g., array, vector, struct, etc.), which makes the implementation of the pass very complex due to the large number of cases requiring special handling.

Instrumentation at execution time instead shows other features:

- ⊕ There is no need to recompile any part of the program.
- ⊕ Modern DBTs expose (almost) architecture-independent IRs, making it easy to support a large set of instructions.
- ⊕ Several high-level concepts are simplified away at the binary level. For instance, there is not much difference between a pointer, an unsigned value, an unsigned struct field, etc.
- ⊖ The instrumentation code may be not very efficient. Tools generate it considering one instruction, or one basic block, at a time. Moreover, DBTs have limited optimization capabilities during the JIT translation. Finally, DBTs may expect

tools to inject branchless code in blocks, forcing executors to inject calls to helpers when conditional actions are desired.

⊖ A DBT keeps track of the original program state, often devising a virtual CPU, whose state is kept in memory, requiring several memory accesses to update it.

⊖ The IR of DBTs may not explicitly model *specialized* instructions. Hence, DBTs may rely on *helper* functions, i.e., hand-written code that manipulates the virtual state to mimic the native instructions. For instance, QEMU uses helpers to model several x86 instructions, such as the division operation, vectorized instructions, and floating-point operations. Executors need to track their effects over the symbolic state. SymQEMU ignores these effects, giving up in terms of accuracy. Other tools [8] exploit models, which, however, are hard to write and easy to get wrong. S2E [19] explores a different strategy: each helper is seen as an additional piece of the analyzed program, translated into the LLVM IR, and interpreted with KLEE to keep the symbolic state consistent.

⊖ Even when an executor can instrument specialized instructions, the resulting expressions may be extremely complex, making it hard for an SMT solver to reason over them.

Optimizations in concolic execution. Modern concolic executors adopt several optimizations that trade accuracy for scalability.

For instance, linearization, or basic block pruning [44], makes the executors track which parts of the program are creating symbolic expressions: when the counter for a code site exceeds a user-defined threshold, the engine returns concrete expressions for that site instead of symbolic expressions. Figure 2 ignores this technique.

A symbolic memory access emerges when the value of a pointer is input-dependent, e.g., when an array index is symbolic. Handling symbolic pointers is hard [5, 9]. To favor scalability, executors concretize symbolic pointers. To still consider alternative values, they may generate a few alternative inputs for each symbolic pointer.

Since concolic executors may visit the same branch conditions several times across different executions, they often keep track of the sites generating branch queries using a bitmap [44], avoiding to repeat queries over time. To make this mechanism less conservative, queries are pruned based on an exponential back-off. Moreover, tools may take into account the calling context of each site.

Hybrid fuzzing. Concolic execution may still struggle to scale over complex programs. Hence, modern concolic executors [8, 36, 37, 44] are often executed in parallel with a coverage-guided fuzzer [24], devising *hybrid fuzzing*. In particular, the concolic executor picks inputs from the queue of the fuzzer and runs over each picked input with a user-defined timeout, e.g., up to 90 seconds, while generating alternative inputs. The fuzzer periodically *imports* inputs from the queue of the concolic executor, accepting only inputs that increase the code coverage (or other features tracked by the fuzzer). Hence, the choice about which inputs are analyzed with concolic execution depends on the choices made by the fuzzer.

Further refinements. Several works [10, 43, 45] have tried to tackle the problem of *path prioritization.* This is crucial as the number of paths increases exponentially in most programs. Thus, several strategies aim at selecting which paths should be analyzed first with symbolic execution. This aspect is still relevant for concolic execution, as the executor may need to select the best input to

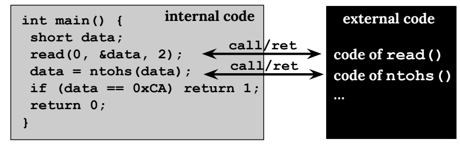


Figure 4: Running example.

explore next. When aiming at coverage, tools may, e.g., pick paths visiting uncovered code [11]. When aiming at bug exploitation, tools may, e.g., favor paths showing security alerts [18]. More in general, a large body of works [2, 13, 17, 33] have contributed to this direction from different perspectives. In this paper, we do not contribute to this problem but we consider a traditional hybrid fuzzing setup, since it the most common setup across recent concolic executors.

3 SYMFUSION

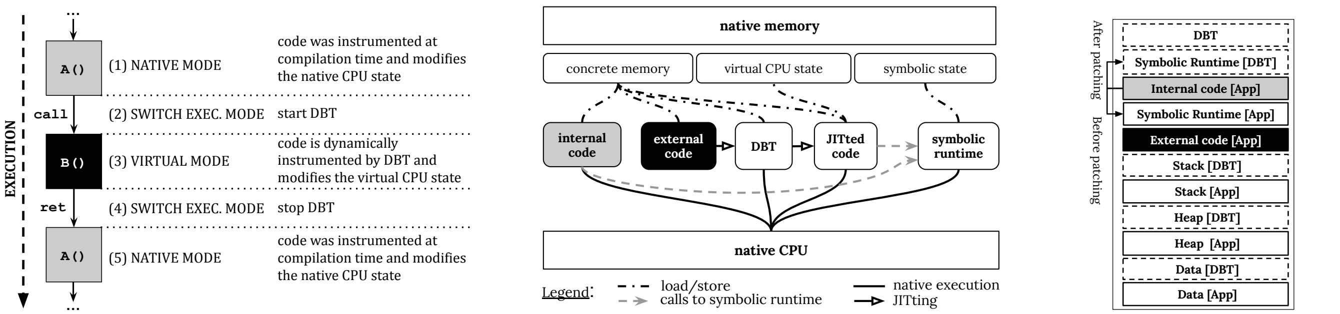
In this section, we present the design of SymFusion.

3.1 Design challenges

The design of SymFusion faces several challenges:

- *Code boundaries.* SymFusion needs to define which part of the program is instrumented at compilation time and which part requires instead instrumentation at execution time.
- *Symbolic state propagation.* The two instrumentation strategies used by SymFusion look at the program from different perspectives (e.g., LLVM IR versus binary code, native execution versus DBT supervised execution, etc.). SymFusion thus needs to intercept when the execution is moving across these perspectives and devise a context switch mechanism able to synchronize and propagate the (symbolic) state.
- *Execution mode of the symbolic runtime.* When SymCC instruments a program, the symbolic runtime becomes one of the dynamic dependencies of the program. Similarly, the DBT of SymQEMU also depends on the symbolic runtime. Hence, when combining these approaches, during the execution, there would be two instances of the symbolic runtime, which are also used in different execution modes (native versus virtual). SymFusion must address this dichotomy to keep the symbolic state always consistent.
- *Function models, or not functions models.* The two instrumentation strategies may exploit function models for different purposes. For instance, SymCC uses them to reason on several C library functions. Conversely, SymQEMU may need them to reason over specialized instructions that are not modeled by the DBT IR. SymFusion should allow the analysis to exploit (good) models when available but then still work accurately when models are missing.

The remainder of this section explains how SymFusion copes with these challenges. To help our discussion, we consider the small program depicted in Figure 4, where 2 bytes are read from the standard input, reversed using ntohs, and then tested in a branch condition.



(a) Main idea. **(b) High-level view of the execution.** **(c) Logical view of the memory.**

Figure 5: How SymFusion works.

3.2 Key ideas

For SymFusion, the code of a program can be conceptually split in:

- *Internal code*, i.e., the code that a user is willing to instrument at compilation time. This covers the core components of the application and, optionally, other dynamic libraries. We color in light gray the code from this category in our figures. In our example, the `main` function is part of the internal code.
- *External code*, i.e., any other dynamic library that a user prefers not to instrument at compilation time. This typically includes system libraries or other third-party libraries that are not typically recompiled by application developers. We color in black the code from this category in our figures. In our example, the C library, which contains the implementation of `read` and `ntohs`, is part of the external code.

Figure 5a depicts the main idea behind SymFusion:

(1) The internal code can run directly on the native CPU. Indeed, its code was generated in order to make calls to the symbolic runtime when the symbolic state requires an update. In our example, `main` can run without any supervision.
(2) When the internal code calls a function of an uninstrumented library (external code), SymFusion requires to intercept this event and perform a *switch* in the execution mode, moving from *native mode* to *virtual mode*. In our example, the switch is performed on the call to `read` and on the call to `ntohs`.
(3) Then, the execution should continue under the supervision of the DBT, which will perform dynamic instrumentation. In our example, the DBT supervises `read` and `ntohs`.
(4) When the function from the external code returns to its caller in the internal code, SymFusion requires to intercept this event and *switch* back the execution mode, moving from *virtual mode* to *native mode*. In our example, the switch is performed when `read` and `ntohs` return to the `main`.
(5) Finally, the internal code should continue its execution on the native CPU, until (2) occurs again or termination.

In a more general sense, SymFusion supports *nested* scenarios where the internal code calls the external code, which in turn calls the internal code, and so on. This scenario happens, e.g., when an application calls the C function `qsort`, which may execute a user-defined *comparator*. Another example is when an application devises a custom wrapper (internal code) around the `malloc` function (external code) and then provides the function pointer of the wrapper to an uninstrumented library (external code).

SymFusion is thus designed to intercept transitions between internal and external code across `call` and `ret` instructions. However, real-world programs may sometimes *break* the `call`/`ret` paradigm when using, e.g., `setjmp`/`longjmp` and other similar primitives. In the next subsection, we provide details about the general execution workflow, while we cover `setjmp`/`longjmp` in Section 4. For the sake of simplicity, we assume that system calls are invoked through external code. Hence, they will be always executed under the supervision of the DBT. Since the internal code can be arbitrarily transformed during compilation, this assumption is not restrictive.

3.3 Execution workflow

Before starting the execution under SymFusion, we expect the user to recompile the internal code of the program using a custom LLVM pass (§3.3.1). Then, the program is ready for concolic execution. Figure 5b provides a high-level view of what would happen:

(1) The host machine can be exemplified in two aspects: the native CPU and the native memory. Any kind of computation from the application should at the end run on the native CPU. Data can be stored either (temporarily) in the native CPU registers or in the native memory. With the term *native CPU state*, we refer to any data stored in the native registers.

(2) The native memory is used to host three main kinds of data: (i) the concrete data of the program, (ii) the virtual CPU state, used by the DBT to keep track of the program CPU state, and (iii) the symbolic state, i.e., the symbolic expressions. During the execution, the concrete memory will also host the code of: the DBT, the symbolic runtime, and the application. Figure 5c shows another view on the memory, where the data are organized based on how they are allocated (stack versus heap versus global data) and based on their owner: analyzed program (solid border) versus DBT (dashed border).

(3) The symbolic runtime is in charge of updating the symbolic state. Hence, any piece of code from the program, regardless if it is from the internal code or the external code, must call the functions of this component to modify the symbolic state. The symbolic runtime devised by SymFusion is an extension of the one used by SymCC and SymQEMU.

(4) When SymFusion is started, the DBT is executed on the native CPU, allowing it to perform its initialization phase. Then, the DBT is used to bootstrap the program execution (§3.3.4), e.g., load into the memory the application code. When done, the native CPU state generated by the DBT is saved and the native execution is diverted at the entry point of the internal code, e.g., starting from the `main` function.

(5) The internal code can now run on the native CPU until a call to the external code is performed. When this event occurs (§3.3.2), the native CPU state is imported into the virtual CPU state, the DBT state is restored and the DBT is restarted.

(6) The DBT then performs JIT translation of the external code, adding calls to the symbolic runtime (§3.3.5). After, the JITted code is executed, allowing the program to make progress.

(7) When the external code calls or returns to the internal code, the native CPU state of the DBT is saved, the virtual CPU state is imported into the native CPU state and then the execution is diverted back to the internal code.

This workflow is repeated until the program's termination. We now review in detail the most interesting aspects of this workflow.

3.3.1 Instrumentation at compilation time. The instrumentation of the internal code is performed using an LLVM pass. As discussed in Section 2, the main idea is to add around each LLVM IR statement one or more calls to the symbolic runtime, guarding them with tests regarding the concreteness of the values manipulated by the statement (§2). Indeed, only when at least one of the operands manipulated by the statement has a symbolic expression associated with it, then there is the need to update the symbolic state.

Our pass is inspired by SymCC, we refer to its paper [36] for more implementation details. However, there are some design choices that characterize SymFusion.

Placement of the pass in the pipeline. Depending on where the pass is inserted within the compilation pipeline, different tradeoffs can be achieved. SymFusion places the pass in the *middle* of the pipeline, i.e., immediately before the LLVM vectorizer (`EP_VectorizerStart`). This allows it to benefit from several optimizations but still process *simple*, i.e., no over-optimized, code. Moreover, SymFusion disables some *destructive* optimizations, e.g., it prevents LLVM from replacing built-in functions with (uninstrumented) inline code. Sanitizers, e.g., ASAN, share this problem but instead prefer to replace built-in functions with ad-hoc wrappers.

Function models. One natural question is whether SymFusion should avoid using function models since uninstrumented code can be tracked using the DBT. While the naive answer is yes, however, in practice, this may lead to worse results. For instance, several functions from the C library are often implemented with vectorized instructions. While SymFusion, as we discuss later, can correctly instrument them, it still may struggle at generating *valuable* symbolic expressions: we will show this in Section 5. Hence, in practice, we do not want to drop completely function models. We can still benefit from them when it makes sense, relying instead on dynamic instrumentation when writing a model is hard or impractical.

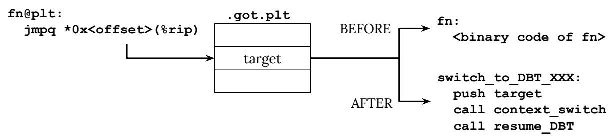
Propagation of the symbolic state. When calling a function, the caller must pass the symbolic expressions associated with the function arguments to the callee. When these functions are part of the internal code, SymFusion, as SymCC, injects calls to specific runtime functions, such as `_sym_set_parameter_expr` in the caller and `_sym_get_parameter_expr` in the callee, to perform such propagation. When the caller and the callee are instead in the external code, SymFusion, as SymQEMU, can easily propagate the symbolic arguments just by working with the shadow registers and the symbolic memory, following the calling convention from the platform ABI. For instance, on Linux `x86_64`, a callee taking one integer argument expects to find the symbolic expression in the shadow register of `RDI`, while a callee taking a floating-point argument expects to find the expression in the shadow register of `XMM0`.

However, when the caller is inside the internal code and the callee inside the external code, or vice versa, it is not clear how to perform such propagation. Indeed, the LLVM pass works on the compiler IR, which is architecture-independent, while the binary code follows platform-dependent rules. SymFusion thus devises inside its DBT a compatibility layer that can still correctly propagate the expressions. To support such an operation, the LLVM pass can help such a layer by making the internal code *export* the knowledge about the number and types of the arguments passed to the callee and the expected return type. For instance, in our example, `main`, before the call to `ntohs`, uses a few functions from the runtime to declare that is passing a single integer argument and is expecting back an integer type. SymFusion can then call `_sym_get_parameter_expr(id=0)` to retrieve the associated expression and map it to the shadow register of `RDI`. Similarly, when `ntohs` returns, SymFusion takes the expression tracked by the shadow register of `RAX` and calls `_sym_set_return_expr`, allowing the internal code to later retrieve the expression with `_sym_get_return_expr`.

Handling indirect calls. SymFusion must know when a callee is part of the external code, since this may require a switch in the execution mode. For direct calls, this can be easily determined statically (§3.3.4). Unfortunately, the same is not true for indirect calls. In these cases, SymFusion is forced to evaluate at execution time whether the target is inside or outside the external code. Hence, the pass replaces each indirect call with a direct call to a *proxy* handler. If the target is within the internal code, the handler jumps to the expected target. Otherwise, it forces a switch to virtual mode. The next section describes how this context switch is performed.

3.3.2 Context switches. To perform the switch between native mode and virtual mode, SymFusion devises several interception mechanisms. We discuss them considering the Linux platform and assume that the external code is composed of dynamic libraries.

Internal code calls external code. For direct calls, the internal code is expected to invoke a stub from the PLT (Procedure Linkage Table). The stub retrieves the actual target address from the GOT (Global Offset Table). The dynamic linker is in charge of populating (resolving) the GOT with the correct addresses: this is done *lazily*, i.e., the first time an entry is needed, or *eagerly* during the process bootstrap phase for all the entries. Hence, the GOT can naturally be exploited to devise a redirection mechanism. In particular, SymFusion forces the eager resolution of the targets, builds a mapping between PLT stubs and their correct targets, and then patches the GOT as depicted by the following figure:



In particular, each target is replaced with the address of a dynamically generated stub, which at running time saves the native CPU state and resumes the DBT execution from the correct virtual PC.

For indirect calls, the internal code invokes the proxy handler, which is in charge of checking whether the target is within the memory boundaries of the external code. When this is true, it performs the same steps of a dynamically generated stub.

Regardless of the call type, while resuming the DBT execution, SymFusion exploits the knowledge on the type and number of arguments (§3.3.1) to propagate the symbolic expressions associated with the arguments, following the rules of the platform ABI.

External code returns to internal code. In this case, SymFusion performs a switch from virtual mode to native mode. There are two possible ways to intercept this event: (a) by monitoring the stack pointer and the instruction pointer in the virtual CPU after `ret` instructions, identifying when, e.g., the stack pointer is within the stack frame of the caller from the internal code, or (b) before executing the external code with the DBT, the return address pushed by the internal code into the stack can be saved into a shadow stack and then replaced with the address of a custom handler, which will thus be executed when the relevant `ret` instruction is executed. For the sake of simplicity, the current implementation favors the second strategy. The custom handler is in charge of performing the context switch, propagating also the symbolic expression of the return value: the expression associated with the shadow register holding the return value (e.g., `RAX` on Linux `x86_64`) is sent to the symbolic runtime, allowing the caller to retrieve it through a dedicated runtime function, such as `_sym_get_return_expr()`.

External code calls internal code. In this case, the DBT monitors the instruction pointer during `call`s, checking when it is falling between the memory boundaries of the internal code. When this happens, a context switch is performed, saving the native CPU state of the DBT and then importing the virtual CPU state into the native CPU state. SymFusion also propagates the symbolic arguments.

Internal code returns to external code. This case can be handled using a similar strategy to what was discussed for the scenario when the external code returns to the internal code.

3.3.3 Execution mode of the symbolic runtime. The internal code is instrumented at compilation time using an LLVM pass that integrates calls to the symbolic runtime. The runtime, however, is not embedded into the program, but it is marked as a dynamic library for the program. The DBT also dynamically depends on the symbolic runtime, as it needs to instrument the external code. This means that, when SymFusion is started, the dynamic loader loads the DBT into memory, loading one copy of the symbolic runtime. The DBT then loads into memory the program: this is done by running the dynamic loader in virtual mode. The dynamic loader in virtual mode loads the internal code, the external code, and another copy of the symbolic runtime. Hence, as shown in Figure 5c, there are two copies of the runtime. However, SymFusion needs to use only one, otherwise, the concolic execution may be inconsistent.

A notable downside of the symbolic runtime linked to the binary is that it shares the stack and the heap with the program. On the other hand, the symbolic runtime linked to the DBT shares the stack and the heap with the DBT. Aiming at isolation, the current implementation favors the symbolic runtime of the DBT. To make the internal code execute the correct copy of the runtime, during the program bootstrap phase (§3.3.4), SymFusion patches the GOT entries associated with functions from the symbolic runtime, forcing the internal code to jump to the correct targets (see Figure 5c). To avoid performing a full and expensive context switch for each call, the symbolic runtime, when called by the internal code, is executed using a hybrid context: it uses the stack of the program but performs dynamic allocations over the heap of the DBT.

3.3.4 Program bootstrap. To execute a program, SymFusion needs to patch the GOT and perform other initialization tasks. These operations require some knowledge about the structure of the program. To this aim, before the program execution, SymFusion statically analyzes the program binary and its dynamic libraries to:

- Identify which dynamic libraries have not been instrumented by the LLVM pass and thus are part of the external code.
- For each component of the internal code, identify the offsets of the GOT entries that need to be patched. These entries can be divided into two categories: targets of the external code and targets of the symbolic runtime. Notice that some GOT entries may be related to libraries that are part of the internal code, which do not require any patching operation.

When running the program under SymFusion, the DBT is executed, which in turn runs the dynamic loader in virtual mode to load the program into memory. When this operation is completed, SymFusion stops the DBT, saves its native CPU state, and performs the patching operations, exploiting the knowledge obtained during the static analysis. Finally, the native execution is diverted into the entry point of the internal code. While the static analysis can be performed only once for each program, the patching operations must be repeated each time the concolic execution is restarted. In Section 4, we discuss how to *amortize* this cost over several runs.

3.3.5 Instrumentation at execution time. To dynamically instrument the external code at execution time, SymFusion extends SymQEMU (see Section 2). One notable limitation of this framework is the lack of symbolic handling for the QEMU helpers. These are extensively used by QEMU to emulate the specialized instructions of a platform which do not have a counterpart in the TCG IR. Examples for the `x86_64` platform are division and remainder operations, vectorized instructions, atomic instructions, and floating-point operations. Moreover, in some cases, QEMU is not able to explicitly model the `eflags` register, thus relying on some helpers.

SymFusion attacks this problem by exploiting its capability of hybrid instrumentation. Indeed, the QEMU helpers are implemented in C code and thus can be easily instrumented at compilation using the LLVM pass and then executed in place of the original helpers. This approach is more general than hand-written symbolic models.

However, since the code instrumented at compilation time is not aware of the virtual CPU, SymFusion has to also inject additional instrumentation code around the helper calls to build a bridge between the two types of instrumentation. This bridge is similar to the compatibility layer discussed in Section 3.3.1.

4 OTHER IMPLEMENTATION DETAILS

A few refinements are needed to make SymFusion effective.

Optimizing the program bootstrap. SymFusion devises also a *shallow* implementation of the symbolic runtime. This can be linked to the internal code in place of the actual runtime implementation. Then, when the dynamic loader executed in virtual mode loads the symbolic runtime into memory, it only needs to load a small executable with no dependencies (while the actual runtime requires several libraries, e.g., the SMT solver), reducing the loader work.

Amortizing the cost of the program bootstrap. Performing the program bootstrap and the patching operations can induce a non-negligible overhead (see Section 5). Hence, SymFusion devises a fork server, which can perform the setup phase only once and then fork the process any time a new concolic execution must be started.

Handling of setjmp and longjmp. A program may use the `setjmp` and `longjmp` primitives to perform non-local `goto`s, breaking the expectations of SymFusion. To overcome this problem, SymFusion dynamically tracks the invocations of these primitives. In particular, when `setjmp` is called, SymFusion saves the passed argument, the current stack pointer, and the current return address. When `longjmp` is called, SymFusion matches the passed argument with one from a previous call to `setjmp`, predicting how the CPU state will be manipulated by the `longjmp` in terms of the instruction pointer and stack pointer after its execution.

Thread-local storage. In Linux `x86_64`, a program can access the Thread-Local Storage (TLS) through the register `fs`. When the program is analyzed by SymFusion, there are two instances of the TLS: one for the original program and one for the DBT. Hence, the `fs` register should be restored when performing a context switch. Unfortunately, updating the `fs` register is expensive: only recent kernel releases allow a program in user mode to update its value, while before a system call was required. Even when this register can be updated in user mode, the cost may be non-negligible as it requires specialized instructions. Hence, SymFusion favors a different strategy: the native CPU always runs with the value of the `fs` register from the DBT even when the internal code is running. To keep the execution consistent, the LLVM pass explicitly adds calls to a few runtime functions that temporarily restore the `fs` register value around the (typically rare) TLS accesses of the internal code.

5 EXPERIMENTAL EVALUATION

In this section, we first consider a few microbenchmarks to validate the design of SymFusion. Then, we evaluate the efficiency and effectiveness of SymFusion on several real-world applications.

5.1 Microbenchmarks

We designed a few programs (M1-6) to validate the design choices behind SymFusion. Table 1 reports the main goal of each program. We now summarize what we experimentally observed:

#	Benchmark description
M1	Program containing the code from Figure 2.
M2	Program containing the code from Figure 3.
M3	Program containing a division operation.
M4	Program using the C library function `ntohl`.
M5	Program using the C library function `strlen`.
M6	Program containing a loop with a nested call to the C library fun. `malloc`.

Table 1: Microbenchmarks [20].

M1: When analyzing a program containing the code from Figure 2, SymFusion can *almost* match the efficiency of SymCC, resulting in a slowdown of 1.02× (versus 6.5× of SymQEMU) thanks to the hybrid instrumentation.

M2: When considering a program containing the code from Figure 3, similarly to SymQEMU and differently from SymCC, SymFusion can track the propagation of symbolic values even when `lib_identity_fn()` is part of the external code.

M3: When running a program with a division operation over symbolic data, SymFusion can build the correct symbolic expression, regardless if the operation is within the internal code or the external code. SymCC can track it only when the operation is inside the internal code. SymQEMU does not track it at all on several platforms, e.g., on `x86_64`, since QEMU may use helpers to handle it but SymQEMU ignores them.

M4: SymFusion, as SymQEMU, can correctly track the symbolic effects of `ntohl` (external code) even without a function model for it. In contrast, SymCC must require a function model. Interestingly, SymCC has a model for `ntohl`, but not for `ntohs`. As a result, SymCC does not handle correctly the code in Figure 4.

M5: SymFusion can always track the effects of `strlen` (external code) but it could struggle at generating *valuable* expressions. Indeed, when a function model is available for it, SymFusion, similarly to SymCC, generates *simple* expressions, which help the solver to generate valuable inputs. However, when the model is not available, the *quality* of the expressions depends on the `strlen` implementation. For instance, when `strlen` uses vectorized instructions, SymFusion can reason over them but these instructions may generate symbolic memory accesses, which are concretized by SymFusion, possibly preventing the generation of valuable inputs. Hence, SymFusion still exploits function models when they are available. SymQEMU instead ignores the effects of vectorized instructions.

M6: Finally, the last program exacerbates one downside of SymFusion: the cost of a context switch. In particular, within each loop iteration, the program switches between external code (a call to `malloc`) and internal code (which just saves the obtained pointer into an array). Hence, for each iteration, two context switches are performed by SymFusion. When the number of iterations is large, e.g., $N = 15000$, SymFusion can be slower than SymCC (11.2×) but also slower than SymQEMU (1.22×). The cost of a context switch could be reduced through the use of specialized instructions. However, in some *extreme* scenarios, its cost is unavoidable.

These microbenchmarks do not always reflect what would happen in a real-world program. Hence, in the next subsection we evaluate SymFusion over more realistic targets.

Program	# inputs	S1: No symbolic expressions			S2: No queries to the solver			S3: Full analysis		
		avg. time (ms)	slowdown wrt SymCC		avg. time (ms)	slowdown wrt SymCC		avg. time (ms)	slowdown wrt SymCC	
		SymCC	SymQEMU	SymFusion	SymCC	SymQEMU	SymFusion	SymCC	SymQEMU	SymFusion
objdump	3687	7	14.4×	3.4×	89	23.4×	1.9×	811	4.0×	3.4×
readelf	8478	4	19.0×	3.0×	41	10.5×	2.4×	641	2.5×	0.3×
tcpdump	5546	4	18.8×	1.8×	23	6.5×	1.1×	398	3.6×	1.0×
libpng	1027	4	51.5×	1.3×	125	16.3×	2.4×	4400	3.3×	1.8×
libtiff	3317	15	9.4×	4.0×	382	12.2×	4.8×	1607	3.1×	4.9×
libxml2	9077	17	11.4×	3.6×	1379	3.5×	1.3×	71461	1.1×	0.9×
php	1591	34	30.5×	5.2×	435	12.2×	4.5×	6166	9.0×	0.7×
poppler	500	800	2.0×	1.3×	2004	5.2×	1.2×	5058	4.2×	1.0×
bsdtar	1753	14	9.7×	9.1×	1083	3.3×	2.2×	4156	1.4×	1.7×
freetype2	7928	8	14.6×	1.3×	278	10.4×	1.5×	5642	3.2×	1.4×
Geo. mean			13.8×	2.7×		8.7×	2.1×		3.0×	1.3×

Table 2: Analysis time when running over the same input queue (generated by AFL in a 2-hour experiment).

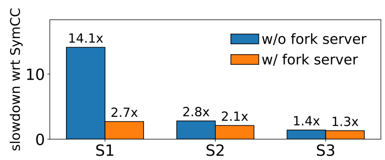


Figure 6(a): Impact of the fork server.

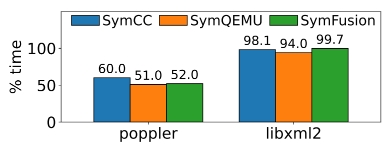


Figure 6(b): Solving time in S3.

Program	S4: Concolic execution			S5: Hybrid fuzzing		
	br. cov.	Δ br. cov. wrt SymCC		br. cov.	Δ br. cov. wrt SymCC	
	SymCC	SymQEMU	SymFusion	SymCC	SymQEMU	SymFusion
objdump	3107	−141	+4	4535	+32	+210
readelf	4123	−607	**+1584**	7094	−393	**+130**
tcpdump	6872	−3109	**+3767**	15991	−1513	**+725**
libpng	1349	+3	**+361**	1847	+32	**+52**
libtiff	1860	−1066	**+2151**	4520	−213	**+22**
libxml2	10079	−330	**+155**	12255	−13	**+143**
php	4451	−114	**+59**	4536	−5	**+10**
poppler	15236	+37	+42	18711	−617	**+387**
bsdtar	2777	−122	+15	3469	−533	+167
freetype2	5420	+181	**+1785**	7770	−599	**+901**
Avg.		−527	+992		−382	+274

Table 6(c): Branch coverage. Delta values for SymFusion are in bold when the Mann–Whitney U test p-value < 0.05 w.r.t. both SymCC and SymQEMU.

5.2 Real-world applications

Evaluating the efficiency and effectiveness of a concolic executor is non-trivial. Indeed, one tool may be faster but inaccurate in building symbolic expressions, while another one may be more accurate but then slower, possibly hitting even more solving timeouts. We thus investigate different scenarios to take into account these tradeoffs.

5.2.1 Experimental setup. We considered 10 applications integrated into the OSS-Fuzz project [27] and often considered by previous works [8, 28, 36, 37]. In particular, we tested: `objdump` and `readelf` from `binutils 2.34`, `bsdtar` (`libarchive`) rev. `f3b1f9`, `freetype2` (`ftfuzzer`) rev. `cd02d3`, `libpng` rev. `a37d483`, `libtiff` (`read_rgba_fuzzer`) rev. `c145a6`, `libxml2` (`read_memory_fuzzer`) rev. `ec6e3e`, `poppler` (`pdf_fuzzer`) rev. `1d2310`, `php` (fuzzer `exif`) rev. `bc39ab`, `tcpdump` 4.9.3 (`pcap` 1.9.1). Regarding which components are part of the internal code, we followed the setup from OSS-Fuzz and Magma [28], using also their seeds. Besides SymFusion, we consider SymQEMU rev. `d1838` and SymCC rev. `9b206`. Each tool ran for 12 hours inside a Docker container based on Ubuntu 20.04, assigning to it 1 core and 4GB of RAM in a server with two Intel Xeon 6238R and 768 GB of RAM. Experiments were repeated 10 times.

5.2.2 Efficiency. To compare the analysis time of different tools, we need to run them exactly on the same input queue. Since the number of seeds for some programs was small, we executed AFL++ 3.14c for 2 hours and then collected its input queue. Table 2 shows the number of inputs for each benchmark and the average running time observed when running SymCC, SymQEMU, and SymFusion. In particular, to help make the comparison, we use SymCC as the baseline and report the slowdown observed with SymQEMU and SymFusion. We considered three experimental scenarios **S1-3**.

In **S1**, we do not inject any symbolic input. Hence, we measure only the running time resulting from having the instrumentation (which does not perform any actual work). We can see that SymCC is the fastest: this is expected as it only uses compile-time instrumentation and tracks only internal code. SymQEMU is 13.8× slower than SymCC. SymFusion can substantially reduce the slowdown, being 2.7× slower than SymCC. This slowdown is expected, as a large amount of time may be spent within the external code and the increased overhead mainly comes from the instrumentation of the external code. Figure 6a sheds light on the benefit of having the fork server: without it, in S1, SymFusion is even slower than SymQEMU. Indeed, the process bootstrap can take a non-negligible amount of time (§3.3.4).

In **S2**, we inject symbolic inputs, allowing a tool to build the expressions, but we disable interactions with the solver. Building expressions increases the work performed by the tools, especially within internal code. As expected, SymCC is still the fastest, followed by SymFusion with a slowdown of 2.1×. Finally, SymQEMU is 8.7× slower than SymCC. Notice that the tools are not doing *exactly* the same work, as they may build different expressions and lose track of data flows in different ways. We also see that with the increase of running time, the cost of the program bootstrap has a lower impact, as shown in Figure 6a.

In **S3**, we consider the full concolic analysis, where a tool can build expressions and submit them to the solver to generate alternative inputs. A tool may be faster and more accurate at generating expressions, but then it may spend a lot of time querying the solver.

For instance, objdump calls the function dcgettext (external code), which generates extremely complex queries. SymCC ignores this function, SymQEMU wrongly generates some of the queries (making them trivially unsolvable), while SymFusion submits several expensive queries, hitting often the solving timeouts (10 seconds for each query). Interestingly, on some programs, SymFusion is faster than SymCC: for instance, on readelf, some external code concretizes one part of the memory but SymCC continues to generate queries when the program works on that part of the memory. Overall, the gap between SymQEMU and SymCC is reduced, which is expected as the solving time plays a crucial role. However, its weight varies across benchmarks. Figure 6b depicts the percentage of time spent in the solver for two programs: in poppler, 51 − 60% of the time is passed in the solver, while, in libxml2, the same percentage is significantly higher. On some benchmarks, such as libxml2 or freetype2, not all tools were able to process the entire queue within the 12 hours: analyzing a single input may take a large amount of time. We also remark that the tools try to avoid repeating queries from the same code site across runs. Hence, the number of queries performed during an experiment depends on the *variety* of inputs: very different inputs may push the executor towards a larger solving time, while similar inputs may reduce the weight of the solving time. Finally, the impact of the fork server in S3 is on average less evident. However, on some programs, such as readelf, it can still help cut the average running time by 20%.

5.2.3 Effectiveness. Following the approach taken by FuzzBench [26], we indirectly evaluate the effectiveness by measuring the code coverage achieved by a program when running over the inputs generated by a specific tool. Table 6c reports the branch coverage measured with gcovr for SymCC. Using SymCC as the baseline, the table also reports the *delta*, i.e., positive or negative increment, on the coverage observed for SymQEMU and SymFusion. We consider two experimental scenarios **S4** and **S5**. In these settings, the input queue of each tool is initialized with the program seeds.

In **S4**, each tool performs a full analysis, as in S3, but also handles its own queue. Hence, while the tools start from the same set of seeds, the input queue over time is affected by the tool's capability of generating new *interesting* inputs. To evaluate whether an input is interesting and should be placed in the queue, tools used afl-showmap (as done in a traditional hybrid fuzzing setup [36, 37, 44]). We can see that SymFusion can, on several programs, significantly increase the code coverage over SymCC. Indeed, although SymFusion is slower than SymCC, it can track the external code, which in turn may lead to the generation of additional *interesting* inputs. In some benchmarks, e.g., bsdtar and objdump, SymFusion generated very complex expressions, hitting the solving timeout more frequently than SymCC: since the exploration over an input is aborted after 90 seconds, SymFusion failed to reach some *deep* branches in a few executions. When considering SymQEMU, SymFusion is faster and potentially more accurate (when taking into account the QEMU helpers), which can give an edge in the long run as the tool can perform more runs and produce more inputs.

In **S5**, we tested the tools in a traditional hybrid fuzzing setup, thus running each tool in parallel with an instance of AFL++. In this setup, the concolic executor picks inputs from the queue of AFL++, while AFL++ periodically imports interesting inputs from the input queue of the concolic executor. AFL++ was used in LLVM mode, thus recompiling the internal code of the programs to maximize the fuzzing effectiveness. The first insight from the results shown in Table 6c is that the S5 setup is able to reach higher code coverage for all programs than what was observed for S4. Interestingly, SymFusion is still able to show a positive delta when compared to SymCC. However, this positive delta is less prominent. After investigating these results, we made three main observations.

First, AFL++ was able to generate several inputs associated with program behaviors that in S4 were exclusively generated by inputs produced by SymFusion. This is not unexpected as modern coverage-guided fuzzers, such as AFL++, have been thoroughly tested over the programs that we considered, achieving extremely high coverage. We remark that we considered S5 and programs from OSS-Fuzz because this is one experimental setup that the research community may consider, e.g., in FuzzBench.

Second, while both SymFusion and SymQEMU may successfully flip branches in the external code, afl-showmap, which is used to evaluate whether an input is interesting, may discard the generated inputs when they only generate new behaviors in the external code. However, when analyzed with a concolic executor, they could lead to the generation of inputs resulting in new behaviors even inside the internal code. Hence, the current hybrid fuzzing setup, used by most recent concolic executors, may *waste* some analysis work.

Third, recent concolic executors, including SymFusion, are not handling symbolic memory accesses. However, reasoning on them may help generate inputs that can be hardly generated by a coverage-guided fuzzers. Unfortunately, adding this capability may significantly slow down the concolic analysis [21]. The state-of-the-art symbolic executor KLEE, which can reason over symbolic accesses, has been shown [32] to struggle at reaching the same coverage obtained by modern fuzzers in FuzzBench. Additionally, handling symbolic memory accesses is even harder when working at binary level, since some knowledge about the program, such as the size of the objects or how the stack frame is organized, is not available.

6 LIMITATIONS

The current implementation of SymFusion has some limitations.

First, the current prototype is not thread-safe. In particular, while the design behind SymFusion can naturally cope even with threads (assuming that the system call clone is executed under the supervision of the DBT), the current implementation of the symbolic runtime is not thread-safe since it exploits several global data structures. SymCC and SymQEMU share the same limitation.

Second, the context switch operation is implemented through inline x86_64 assembly code to minimize the overhead. Hence, this code must be revised when porting SymFusion to other platforms.

Third, while SymFusion could instrument the floating-point instructions, both in the LLVM pass and inside the DBT, its symbolic runtime does not yet know how to generate the correct expressions in the SMT solver. SymCC and SymQEMU share the same limitation.

Fourth, the current prototype does not handle the stack unwinding operation performed by some C++ exception handlers. In particular, similarly to longjmp, these handlers may break the call/return paradigm expected by SymFusion. To properly handle stack unwinding, SymFusion should closely monitor the stack and instruction pointer during the execution to understand how the program is unraveling the stack frames.

Finally, the LLVM pass does not force a switch to the virtual mode when some inline assembly code is met within a function. SymFusion also assumes that static libraries are always part of the internal code, ignoring the scenario where a program statically links a library that has not been instrumented with the LLVM pass. To handle such a case, SymFusion should force a switch to the virtual mode in presence of calls to the uninstrumented static library.

7 CONCLUSIONS

This paper presents SymFusion, a novel design for a concolic executor based on *hybrid* instrumentation.

From one side, compilation time instrumentation allows a concolic executor to track the effects of a piece of code with small overhead but requires recompilation, which can be tricky in presence of third-party components, such as system libraries. On the other side, instrumentation at execution time performed, e.g., with a dynamic binary translator (DBT), does not require to recompile the program but may generate less efficient instrumented code, increasing the analysis overhead. Unfortunately, when the code is not instrumented, the analysis may generate inaccurate expressions.

SymFusion allows the user to instrument only the core components of an application at compilation time with an LLVM pass but then still tracks the effects of the remaining code using a DBT. Our experiments show that SymFusion can provide a nice balance between accuracy and efficiency when analyzing complex real-world applications. Indeed, SymFusion, when compared to SymCC and SymQEMU, is able to achieve higher code coverage over time.

As future work, we identify four main directions. First, we plan to improve the symbolic runtime, making it thread-safe and adding support for floating-point operations. Second, accurate handling of symbolic memory accesses could make SymFusion more effective. Third, the LLVM pass could be extended to force a switch in virtual mode when inline assembly code is met within a function. Finally, we believe that the traditional hybrid fuzzing setup is limiting the potential behind SymFusion and alternative setups should be thus explored. Several existing works [43, 45] on this research topic did not consider modern concolic executors, hence, it is not clear how they would perform without an extended experimental evaluation.

ACKNOWLEDGMENTS

This work is supported, in part, by the National Science Foundation under Grant No. 2133487. Any opinions, findings, conclusions, or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding agencies.

REFERENCES

[1] Thanassis Avgerinos, Sang Kil Cha, Alexandre Rebert, Edward J. Schwartz, Maverick Woo, and David Brumley. 2014. Automatic Exploit Generation. *Commun. ACM* 57, 2 (feb 2014), 74–84. https://doi.org/10.1145/2560217.2560219

[2] Roberto Baldoni, Emilio Coppa, Daniele Cono D'Elia, Camil Demetrescu, and Irene Finocchi. 2018. A Survey of Symbolic Execution Techniques. *ACM Computer Surveys* 51, 3, Article 50 (2018). https://doi.org/10.1145/3182657

[3] Thomas Ball, Ella Bounimova, Byron Cook, Vladimir Levin, Jakob Lichtenberg, Con McGarvey, Bohus Ondrusek, Sriram K. Rajamani, and Abdullah Ustuner. 2006. Thorough Static Analysis of Device Drivers. In *Proc. 1st ACM SIGOPS/EuroSys European Conf. on Comp. Systems (EuroSys'06)*. ACM, 73–85. https://doi.org/10.1145/1217935.1217943

[4] Fabrice Bellard. 2005. QEMU, a Fast and Portable Dynamic Translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference (ATEC '05)*. USENIX Association, Berkeley, CA, USA, 41–41. http://dl.acm.org/citation.cfm?id=1247360.1247401

[5] Luca Borzacchiello, Emilio Coppa, Daniele Cono D'Elia, and Camil Demetrescu. 2019. Memory models in symbolic execution: key ideas and new thoughts. *Software Testing, Verification and Reliability* 29, 8 (2019). https://doi.org/10.1002/stvr.1722

[6] Luca Borzacchiello, Emilio Coppa, Daniele Cono D'Elia, and Camil Demetrescu. 2019. Reconstructing C2 Servers for Remote Access Trojans with Symbolic Execution. In *Cyber Security Cryptography and Machine Learning (CSCML '19)*. Springer International Publishing. https://doi.org/10.1007/978-3-030-20951-3_12

[7] Luca Borzacchiello, Emilio Coppa, and Camil Demetrescu. 2021. Fuzzing Symbolic Expressions. In *Proceedings of the 43rd International Conference on Software Engineering (ICSE '21)*. https://doi.org/10.1109/ICSE43902.2021.00071

[8] Luca Borzacchiello, Emilio Coppa, and Camil Demetrescu. 2021. FUZZOLIC: mixing fuzzing and concolic execution. *Computers & Security* (2021). https://doi.org/10.1016/j.cose.2021.102368

[9] Luca Borzacchiello, Emilio Coppa, and Camil Demetrescu. 2022. Handling Memory-Intensive Operations in Symbolic Execution. In *Proceedings of the 15th Innovations in Software Engineering Conference (ISEC '22)*. https://doi.org/10.1145/3511430.3511453

[10] Jacob Burnim and Koushik Sen. 2008. Heuristics for Scalable Dynamic Test Generation. In *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*. 443–446. https://doi.org/10.1109/ASE.2008.69

[11] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI'08)*. http://dl.acm.org/citation.cfm?id=1855741.1855756

[12] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. 2008. EXE: Automatically Generating Inputs of Death. *ACM Trans. Inf. Syst. Secur.* 12, 2, Article 10 (2008). https://doi.org/10.1145/1455518.1455522

[13] Cristian Cadar and Koushik Sen. 2013. Symbolic Execution for Software Testing: Three Decades Later. *Commun. ACM* 56, 2 (2013), 82–90. https://doi.org/10.1145/2408776.2408795

[14] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. 2012. Unleashing Mayhem on Binary Code. In *Proc. 2012 IEEE Symp. on Sec. and Privacy (SP'12)*. IEEE Comp. Society, 380–394. https://doi.org/10.1109/SP.2012.31

[15] Ju Chen, Wookhyun Han, Mingjun Yin, Haochen Zeng, Chengyu Song, Byoungyoung Lee, Heng Yin, and Insik Shin. 2022. SymSan: Time and Space Efficient Concolic Execution via Dynamic Data-Flow Analysis. In *USENIX Security Symposium (Security)*. USENIX Association. https://www.usenix.org/conference/usenixsecurity22/presentation/chen-ju

[16] Ju Chen, Jinghan Wang, Chengyu Song, and Heng Yin. 2022. JIGSAW: Efficient and Scalable Path Constraints Fuzzing. In *Proceedings of the 43rd IEEE Symposium on Security and Privacy (SP '22)*. https://doi.org/10.1109/SP46214.2022.9833796

[17] Yaohui Chen, Mansour Ahmadi, Reza Mirzazade farkhani, Boyu Wang, and Long Lu. 2020. MEUZZ: Smart Seed Scheduling for Hybrid Fuzzing. In *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*. 77–92. https://www.usenix.org/conference/raid2020/presentation/chen

[18] Yaohui Chen, Peng Li, Jun Xu, Shengjian Guo, Rundong Zhou, Yulong Zhang, Tao Wei, and Long Lu. 2020. SAVIOR: Towards Bug-Driven Hybrid Testing. In *2020 IEEE Symposium on Security and Privacy (SP)*. 1580–1596. https://doi.org/10.1109/SP40000.2020.00002

[19] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. 2012. The S2E Platform: Design, Implementation, and Applications. *ACM Trans. on Computer Systems (TOCS)* 30, 1 (2012), 2:1–2:49. https://doi.org/10.1145/2110356.2110358

[20] Emilio Coppa. 2022. SymFusion repository. https://season-lab.github.io/SymFusion/

[21] Emilio Coppa, Daniele Cono D'Elia, and Camil Demetrescu. 2017. Rethinking Pointer Reasoning in Symbolic Execution. In *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering (ASE '17)*. https://doi.org/10.1109/ASE.2017.8115671

[22] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Proceedings of 14th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08/ETAPS'08)*. https://doi.org/10.1007/978-3-540-78800-3_24

[23] Sushant Dinesh, Nathan Burow, Dongyan Xu, and Mathias Payer. 2020. RetroWrite: Statically Instrumenting COTS Binaries for Fuzzing and Sanitization. In *2020 IEEE Symposium on Security and Privacy (SP)*. https://doi.org/10.1109/SP40000.2020.00009

[24] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. AFL++: Combining incremental steps of fuzzing research. In *Proceedings of the 14th USENIX Workshop on Offensive Technologies (WOOT '20)*.

[25] Patrice Godefroid, Michael Y. Levin, and David A. Molnar. 2008. Automated Whitebox Fuzz Testing. In *Proc. Network and Distributed System Security Symp. (NDSS'08)*.

[26] Google. 2022. FuzzBench. https://github.com/google/fuzzbench/.

[27] Google. 2022. Google OSS-Fuzz: continuous fuzzing of open source software. https://github.com/google/oss-fuzz.

[28] Ahmad Hazimeh, Adrian Herrera, and Mathias Payer. 2020. Magma: A Ground-Truth Fuzzing Benchmark. *Proc. ACM Meas. Anal. Comput. Syst.* 4, 3, Article 49 (Dec. 2020), 29 pages. https://doi.org/10.1145/3428334

[29] J.C. Huang. 1978. Program Instrumentation and Software Testing. *Computer* 11, 4 (1978), 25–32. https://doi.org/10.1109/C-M.1978.218134

[30] James C. King. 1976. Symbolic Execution and Program Testing. *Commun. ACM* 19, 7 (1976), 385–394. https://doi.org/10.1145/360248.360252

[31] C. Lattner and V. Adve. 2004. LLVM: a compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* 75–86. https://doi.org/10.1109/CGO.2004.1281665

[32] Jonathan Metzman Laurent Simon, Read Sprabery. 2021. Klee in FuzzBench. https://srg.doc.ic.ac.uk/klee21/talks/Simon-FuzzBench.pdf.

[33] Dongge Liu, Gidon Ernst, Toby Murray, and Benjamin I. P. Rubinstein. 2020. Legion: Best-First Concolic Testing. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering (ASE '20).* https://doi.org/10.1145/3324884.3416629

[34] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '05).* https://doi.org/10.1145/1065010.1065034

[35] Sebastian Poeplau and Aurélien Francillon. 2019. Systematic Comparison of Symbolic Execution Systems: Intermediate Representation and Its Generation. In *Proceedings of the 35th Annual Computer Security Applications Conference (ACSAC '19).* https://doi.org/10.1145/3359789.3359796

[36] Sebastian Poeplau and Aurélien Francillon. 2020. Symbolic execution with SymCC: Don't interpret, compile!. In *29th USENIX Security Symposium (USENIX Security 20).* https://www.usenix.org/system/files/sec20-poeplau.pdf

[37] Sebastian Poeplau and Aurélien Francillon. 2021. SymQEMU: Compilation-based symbolic execution for binaries. In *Network and Distributed System Security Symposium.* Network & Distributed System Security Symposium.

[38] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. 2010. All You Ever Wanted to Know About Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask). In *Proceedings of the 2010 IEEE Symposium on Security and Privacy (SP 2010).* 317–331. https://doi.org/10.1109/SP.2010.26

[39] Koushik Sen, Darko Marinov, and Gul Agha. 2005. CUTE: A Concolic Unit Testing Engine for C. In *Proc. 10th European Software Engineering Conf. Held Jointly with 13th ACM SIGSOFT Int. Symp. on Foundations of Software Engineering (ESEC/FSE'13).* 263–272. https://doi.org/10.1145/1081706.1081750

[40] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker. In *2012 USENIX Annual Technical Conference (USENIX ATC 12).* 309–318.

[41] Yan Shoshitaishvili, Ruoyu Wang, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2015. Firmalice - Automatic Detection of Authentication Bypass Vulnerabilities in Binary Firmware. In *22nd Annual Network and Distributed System Security Symp. (NDSS'15).* https://doi.org/10.14722/ndss.2015.23294

[42] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2016. SOK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *IEEE Symp. on Security and Privacy (SP'16).* 138–157. https://doi.org/10.1109/SP.2016.17

[43] Xinyu Wang, Jun Sun, Zhenbang Chen, Peixin Zhang, Jingyi Wang, and Yun Lin. 2018. Towards Optimal Concolic Testing. In *Proceedings of the 40th International Conference on Software Engineering (ICSE '18).* 291–302. https://doi.org/10.1145/3180155.3180177

[44] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. 2018. QSYM: A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing. In *27th USENIX Security Symposium (USENIX Security 18).* USENIX Association. https://www.usenix.org/conference/usenixsecurity18/presentation/yun

[45] Lei Zhao, Yue Duan, Heng Yin, and Jifeng Xuan. 2019. Send Hardest Problems My Way: Probabilistic Path Prioritization for Hybrid Fuzzing. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019.* https://www.ndss-symposium.org/ndss-paper/send-hardest-problems-my-way-probabilistic-path-prioritization-for-hybrid-fuzzing/