SymFit: Making the Common (Concrete) Case Fast for Binary-Code Concolic Execution

Zhenxiao Qi, Jie Hu, Zhaoqi Xiao, and Heng Yin, *UC Riverside*

https://www.usenix.org/conference/usenixsecurity24/presentation/qi

This paper is included in the Proceedings of the
33rd USENIX Security Symposium.

August 14–16, 2024 • Philadelphia, PA, USA

978-1-939133-44-1

SymFit: Making the Common (Concrete) Case Fast for Binary-Code Concolic Execution

Zhenxiao Qi
UC Riverside

Jie Hu
UC Riverside

Zhaoqi Xiao
UC Rvierside

Heng Yin
UC Rvierside

Abstract

Concolic execution is a powerful technique in software testing, as it can systematically explore the code paths and is capable of traversing complex branches. It combines concrete execution for environment modeling and symbolic execution for path exploration. While significant research efforts in concolic execution have been directed toward the improvement of symbolic execution and constraint solving, our study pivots toward the often overlooked yet most common aspect: concrete execution. Our analysis shows that state-of-the-art binary concolic executors have largely overlooked the overhead in the execution of concrete instructions. In light of this observation, we propose optimizations to make the common (concrete) case fast. To validate this idea, we develop the prototype, SymFit, and evaluate it on standard benchmarks and real-world applications. The results showed that the performance of pure concrete execution is much faster than the baseline SymQEMU, and is comparable to the vanilla QEMU. Moreover, we showed that the fast symbolic tracing capability of SymFit can significantly improve the efficiency of crash deduplication.

1 Introduction

Symbolic execution [9, 13, 16, 26, 27] is a powerful technique for automated software testing that has gained significant attention in recent years. It has been widely used in finding security vulnerabilities due to its capability of effective code path exploration. Despite its effectiveness, symbolic execution is also known to be expensive. Recently, the efficiency of concolic execution has been greatly improved. SymCC [24] and SymSan [11] take a compile-time instrumentation approach to collect and solve path constraints and are several orders of magnitude faster than QSYM [31], which is based on dynamic binary instrumentation. SymSan [11] is even faster than SymCC [24] by utilizing the highly optimized shadow memory from data-flow sanitizer [30]. While SymCC and SymSan require the source code of the target program to perform concolic execution, QSYM [31] and SymQEMU [23] provide a binary-only solution by directly weaving the concolic execution logic into the dynamic binary translation process.

Essentially, concolic execution combines concrete and symbolic execution. While significant research efforts in concolic execution have been directed toward the improvement of symbolic execution and constraint solving [11, 12, 23, 31], our study pivots toward the often overlooked yet most common aspect, concrete execution. First, we review the design of existing binary concolic executors, focusing on how concrete and symbolic execution are mixed, and evaluate the overhead of handling concrete execution in the state-of-the-art binary concolic executor, SymQEMU. We observed that even though SymQEMU has significantly improved the performance of concolic execution on binary code over QSYM, it still exhibits a large overhead in handling the execution of concrete instructions. Our evaluation shows that when no symbolic inputs are introduced, the concrete execution in SymQEMU is 35× slower than vanilla user-mode QEMU on SPEC CINT-2006 benchmarks.

"Optimize for the common case" is a principle in computer science and engineering that suggests designing and optimizing systems, algorithms, and software based on the most frequent or expected use cases, rather than focusing primarily on edge cases or rare scenarios. Inspired by this principle, we design an efficient concolic execution framework with optimization for the common case, concrete execution. We note that in concrete execution, only load and store instructions need to be checked as they can potentially access symbolic values in the shadow memory, the rest instructions can run natively without instrumentation. To reduce the latency of checking concrete shadow memory, we designed a Concrete Memory Lookaside Buffer (CMLB) that caches recent-accessed concrete memory pages. As a result, checking whether the address is concrete can be performed by a few instructions that look up CMLB. When loading a symbolic value in concrete mode, the execution should transit to symbolic mode, where instructions are monitored to collect their symbolic effects. More importantly, the concolic execution engine automatically switches between concrete mode and symbolic mode to avoid unnecessary monitoring of concrete instructions.

We build our prototype, SymFit, on top of user-mode QEMU, with our optimizations for the common (concrete) cases. For the frontend, we follow the instrumentation strategy in SymQEMU to propagate and collect symbolic constraints. For the backend, we adopt the ideas in SymSan [11] for its efficient symbolic state and shadow memory management.

We compared SymFit with its baseline SymQEMU, and found that the overhead in concrete execution is greatly reduced on standard benchmarks (SPEC CINT-2006 [5] and Linux/Unix nbench [3]) and real-world applications (Google's Fuzzbench [2] and Unibench, the real-world benchmark presented in UniFuzz [20]). The evaluation results showed that compared to SymQEMU, SymFit achieved 13× and 20× speedup on SPEC CINT and nbench for pure concrete execution. When the overhead of constraint solving is excluded, SymFit is 12.03× faster than SymQEMU on Fuzzbench programs. When solving is enabled, SymFit can still enjoy a 6.67× speedup over SymQEMU. In the end-to-end hybrid fuzzing experiment, SymFit can achieve similar edge coverage faster than SymQEMU and sometimes achieve higher edge coverage. Moreover, the fast symbolic tracing capability of SymFit can significantly improve the efficiency of bug detection and crash deduplication.

Contributions. In summary, we make the following contributions:

- We review the system designs of existing binary concolic executors, with respect to how they combine concrete and symbolic executions. We make an observation that these designs incur significant overhead for concrete execution, which is the common case.
- We propose two key optimizations, lightweight concrete mode and concrete memory lookaside buffer, which significantly reduce the concrete execution overhead.
- We build a prototype, SymFit, and evaluate it on standard benchmarks (SPEC-CINT and nbench), real-world applications (Fuzzbench and Unifuzz), and security applications (crash deduplication). Moreover, we make SymFit publicly available to foster future research in the area.

2 Background and Motivation

Symbolic execution is a technique that can explore all possible execution paths in software using symbolic values instead of concrete inputs. It tracks the values of symbolic inputs as they are propagated by the program, and maintains a map from program variables (including registers and memory) to their symbolic expressions. When the engine encounters a branch statement with symbolic operands, it constructs a boolean formula based on collected symbolic expressions and checks the feasibility using an SMT solver. To generate a concrete input that follows the same execution path, the engine queries the SMT solver for feasible assignments to symbolic values.

Concolic Execution. One challenge of classic symbolic execution is that it fails to explore the execution path when the path constraints are too complex for the SMT solver to solve within a limited time. To alleviate this issue, researchers have proposed concolic execution, where the symbolic path exploration is guided by the execution of the program with concrete inputs. To explore executions that deviate from the current concrete execution path, the concolic execution engine checks the feasibility of the opposite branch by querying the SMT solver and generates concrete inputs leading to that direction if feasible.

2.1 Source-Code Concolic Execution

In order to trace the execution of the target program, symbolic execution engines need to understand the underlying instruction set. Classic symbolic execution engines [9,13,28] perform the analysis on the intermediate representation (IR) lifted from the program. At the core of such engines, an interpreter runs the IR symbolically and keeps a record of how symbolic values are computed in the program. While IR interpretation-based approaches are portable and architecture-agnostic, they all face scalability issues. Recently, a line of research works has shown that compiling or instrumenting the symbolic execution logic into the program can benefit from the native execution of the program, thus much faster than IR interpretation-based approaches. For example, SymCC [24] and SymSan [11] compile symbolic execution logic into the target program at LLVM IR level. They hook into the compiler and inject library function calls that interact with the symbolic execution backend. Such approaches also benefit from compiler optimizations, and instrumentation only needs to be done once for each program. However, compilation-based approaches fundamentally require a compiler, thus not applicable when the source code of the program or third-party libraries are not accessible.

2.2 Binary-Code Concolic Execution

While source-code concolic execution (such as SymCC [24] and SymSan [11]) is highly efficient, it has drawbacks too. The source code of the target program as well as all the dependent libraries must be available for compiler instrumentation. This assumption does not hold for proprietary software and library components. Even if the source code of all the software components is available, one may still encounter various compatibility issues when compiling these components with the compilers that come with these concolic execution engines. If for any reason a component cannot be properly compiled, function summaries must be provided for the functions exported by this component. Otherwise, constraint collection might be incomplete or incorrect. However, writing function summaries can be a tedious and error-prone process by itself. Therefore, concolic execution that can directly work on binary code is desirable.

In essence, concolic execution combines symbolic execution and concrete execution to explore program execution space. In this subsection, we review the system designs of existing binary-code concolic executors, with respect to how they combine concrete and symbolic executions, as illustrated in Figure 1.

Angr. Angr [17, 28] is a binary concolic executor that utilizes the Unicorn engine [25] (i.e. user-mode QEMU) for concrete execution and the Valgrind framework for symbolic interpretation. To avoid heavyweight symbolic interpretation, Angr provides APIs for users to select functions or code regions for symbolic exploration (Figure 1). Initially, Angr executes the target binary concretely via the Unicorn engine to aggregate a concrete environment model, i.e., concrete values for registers and memory. Upon reaching a user-defined *Point of Interest (PoI)*, a symbolic state is initialized, replicating the concrete states and setting user-selected states (register or memory) as symbolic. Then symbolic execution is performed with initialized symbolic states. Once the symbolic exploration converges on the *Target Point (TP)*, symbolic execution terminates and each symbolic variable is assigned a concrete value that satisfies collected symbolic constraints and synchronized with concrete states to drive the concrete execution.

S2E. S2E [13] runs the entire operating system in the whole-system emulator QEMU and connects to KLEE [9] to symbolically execute selected code regions (Figure 1). Similar to Angr's strategy, it minimizes symbolic emulation overhead by letting users pinpoint functions or code regions to explore symbolically. Specifically, it provides two plugins, *Annotation*, and *CodeSelector*, allowing users to annotate variables as symbolic and specify code ranges to perform symbolic execution.

Mayhem. Instead of manually selecting boundaries between symbolic and concrete executions, Mayhem [10] provides a different strategy to reduce unnecessary emulation for concrete instructions with taint tracking. Specifically, the concrete execution engine instruments the target binary and performs dynamic taint analysis to track attacker-controlled input from environment variables, files, and networks. During the concrete execution, tainted instruction traces are streamed to the symbolic executor (KLEE) for symbolic exploration and exploit generation.

Problem 1: Concolic executors like Angr and S2E determine boundaries between concrete execution and symbolic execution upon user selection. This strategy effectively reduces the total overhead of symbolic emulation, as only selected code regions are executed symbolically. However, this strategy is coarse-grained. First, it requires preliminary reverse engineering efforts to first identify target functions or relevant code ranges, and designate which values should be treated as symbolic. Second, even within code segments selected for symbolic execution, many instructions that do not operate on symbolic values should be executed concretely. Third, selected code regions might overlook instructions outside of the selected range that require symbolic emulation. Mayhem performs dynamic taint analysis to ensure tainted instructions are subject to symbolic emulation, while untainted ones are spared. However, dynamic taint analysis introduces additional overhead at runtime, which becomes particularly problematic for whole-system emulation.

Problem 2: Notably, the above approaches all perform symbolic and concrete executions in two processes. As a result, Angr and S2E inevitably need to perform context switches between two engines and non-trivial synchronization between the symbolic and concrete environment, while Mayhem has to lift the target binary twice, one in Pin to perform dynamic taint tracking, another one in BAP for symbolic emulation.

QSYM and SymQEMU. Instrumentation-based approaches, such as QSYM and SymQEMU, were proposed to obviate the need for context switches and synchronization between the symbolic and concrete environment. This is achieved by injecting symbolic emulation logic directly into the target program, which enables running concrete code and symbolic emulation within one process, rendering mode switches extremely lightweight - essentially, a mere function call. Moreover, symbolic emulation logic is instrumented and executed together with the target binary on CPU, yielding a much higher performance compared to interpretation-based symbolic emulation like Angr, S2E, and Mayhem.

To avoid symbolic emulation overhead for concrete instructions, QSYM employs dynamic taint tracking to identify instructions that need symbolic emulation (Figure 1). Specifically, QSYM has to disassemble each instruction at run time and check each operand to determine if this instruction needs to be executed concretely or symbolically. The disassembling and taint checking inevitably introduce additional overhead. Notably, symbolic emulation intrinsically propagates symbolic data. Therefore, SymQEMU (Figure 1), rather than relying on additional taint tracking, evaluates the need for symbolic emulation for an instruction by checking if its operand(s) (variables or memory addresses) are symbolic. This method enables it to bypass symbolic emulation for concrete instructions while eliminating the need for dynamic taint tracking and shows better efficiency than QSYM.

2.3 The Common Case Overhead

Our study pivots towards the often overlooked yet most common aspect: concrete execution, rather than symbolic execution and constraint solving methodologies, where significant research efforts have been directed towards [11, 12, 23, 31].

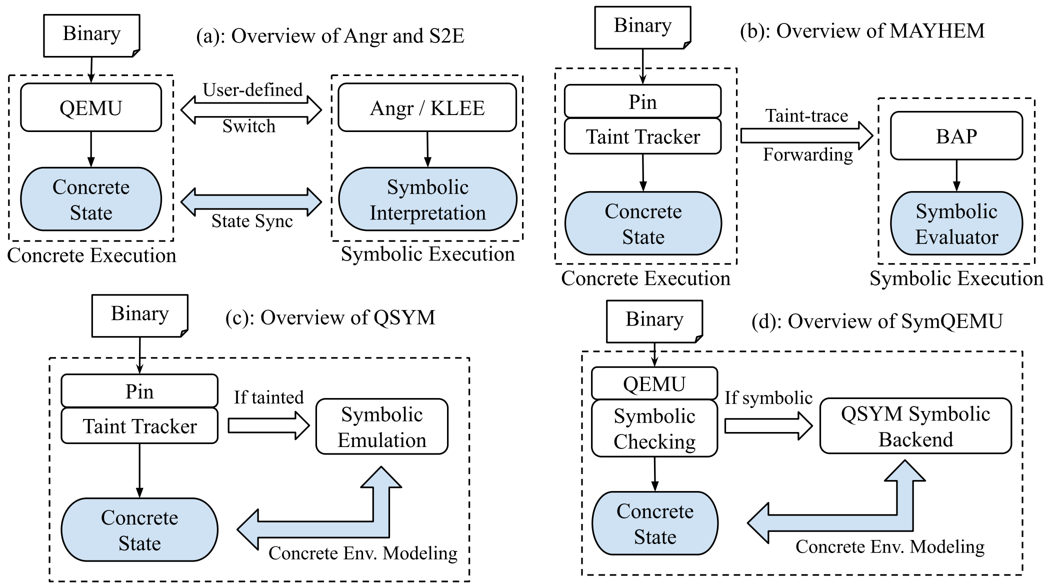

Figure 1: How concrete execution and symbolic execution are combined in binary-code concolic execution

Table 1: SYMQEMU on CINT-2006 w/ concrete inputs

CINT-2006	QEMU (s)	SYMQEMU (s)	Overhead (×)
hmmer	8.00	293.39	36.63
libquantum	541.78	17035.43	31.44
bzip2	179.81	4625.94	25.73
mcf	244.83	3072.34	12.55
sjeng	978.06	29495.09	30.16
gcc	71.82	1749.03	24.35
xalancbmk	308.25	68874.56	223.44
Geo. Mean			34.74

```
1  static inline void tcg_gen_add_i64(TCGv_i64 ret,
2          TCGv_i64 arg1, TCGv_i64 arg2)
3  {
4    // Add a helper function call to IR.
5    tcg_gen_helper_sym_add(ret_expr, arg1_expr, arg2_expr);
6    tcg_gen_op3_i64(INDEX_op_add_i64, ret, arg1, arg2);
7  }
8  void *helper_sym_add(ret_expr, arg1_expr, arg2_expr)
9  {
10   // Concreteness checking.
11   if (arg1_expr == NULL && arg2_expr == NULL) {
12     // Return if both expressions are concrete.
13     return NULL;
14   }
15   ......
16   // Build symbolic expression for add.
17   return _sym_build_add_i64(arg1_expr, arg2_expr);
18 }
```

Listing 1: An example of SYMQEMU's instrumentation using helper functions.

Notably, within the mix of concolic execution, concrete execution often emerges as the "common case". In this section, we want to gain insights into how good SYMQEMU, the state-of-the-art binary concolic, handles the common case, concrete execution. We conducted a preliminary evaluation using the SPEC CINT-2006 benchmark programs. Since QEMU fails to run new versions of SPEC CINT benchmarks due to unsupported instructions, only a few benchmarks were selected from SPEC CINT-2006 as motivating examples. By comparing the execution time of pure concrete inputs, we were able to identify the overhead introduced by SYMQEMU in handling concrete execution, in which it only needs to perform concreteness checks on shadow variables and memory. The results (Table 1) indicate an overhead of up to 223× and an average of 35×. According to the evaluation in Mayhem [10] and our observation, the majority of instructions (90%) are executed concretely. Therefore, the room for further performance improvement is still large.

We further analyzed SYMQEMU to understand the overhead of handling the common case, concrete execution. First, to determine whether an instruction should be executed concretely or symbolically, SYMQEMU inserts a helper function call before each instruction (except for `mov`-like instructions) to check whether operands are concrete. If so, the symbolic emulation of this instruction can be skipped. As an example shown in Listing 1, a helper function is added (at line 5) when generating the IR for the `add` instruction. This helper function (at line 8) checks the concreteness (line 11) of operands and builds symbolic expression (line 17) if needed. While checking the concreteness of operands is necessary, doing so by checking every instruction incurs a non-negligible overhead. Secondly, shadow memory checking in SymQEMU is expensive. To check the concreteness of an address, the virtual address is first translated to its corresponding shadow address using a two-level mapping. Then, the shadow address is accessed to determine if it is concrete. As memory accesses are frequent, such overhead can be significant when analyzing real-world applications.

Scope. In this work, we follow the system design principle of "optimize for the common case" to design a concolic executor with optimized concrete execution. While this principle is generic to be applied to different solutions of concolic executors, such as QSYM, our concolic executor follows the design of SymQEMU, as the underlying framework, QEMU, is open-sourced, allowing us to implement our optimization strategies at the system level. We have made key observations of SymQEMU that guide our approach:

- The concrete execution exhibits non-negligible overhead due to concreteness checking at the instruction level. We argue that, in concrete execution, only load and store need to be monitored as they may potentially access shadow memory from memory. No concreteness checking is necessary for the rest instructions to keep concrete execution fast. The transition to symbolic execution happens as soon as a symbolic value is loaded from shadow memory to ensure the correctness of symbolic emulation. In symbolic execution, the engine should switch back to concrete execution as soon as no symbolic emulation is needed.
- (Concrete) shadow memory access can be frequent and imposes significant overhead. Checking for shadow memory can be optimized by caching recent-accessed concrete memory pages. By doing so, the overhead of shadow address translation and shadow memory access can be avoided if the cache hits.

3 System Overview

Essentially, the design of concolic executors consists of the front end for symbolic state collection and the back end for symbolic state management. SymFit follows the front-end design of SymQEMU, which adopts user-mode QEMU for its reasonable performance and multi-architecture support. Instead of reusing the symbolic backend from QYSM as SymQEMU does, we adopt the symbolic state management from SymSan [11], a recent work that shows efficient symbolic expression and shadow memory management. While SymSan requires the source code of target programs, we introduce more details about how we apply SymSan's ideas for binary concolic execution in §3.2.

More importantly, SymFit is designed to optimize for the common (concrete) case. It features a lightweight concrete execution and the ability to efficiently discern the boundary between symbolic and concrete execution. To optimize for the common (concrete) case, it only monitors load and store instructions in concrete execution. When a symbolic load is detected, SymFit switches to the symbolic mode with full instrumentation. When symbolic emulation is no longer necessary, it transitions back to the concrete mode. To determine when to switch, SymFit evaluates the concreteness of CPU registers at the end of each basic block, instead of checking every instruction which has a non-negligible overhead.

Figure 2 illustrates the architecture of SymFit. It is built atop a dynamic binary translator, QEMU, and takes a binary as input. The concolic execution logic is instrumented into the binary dynamically, allowing it to switch between lightweight concrete mode and symbolic execution mode at runtime. The latency of shadow memory access is improved by the Concrete Memory Lookaside Buffer (CMLB) lookup, which is done by a few instructions inlined into native code during translation. SymFit utilizes a symbolic backend to construct symbolic path constraints, query the SMT solver, and generate new test inputs for path exploration.

Lightweight Concrete Execution. As shown in Table 1, even with no symbolic inputs, SymQEMU still incurs a 35 times slowdown compared to the vanilla QEMU. This overhead comes solely from instrumented helper function calls (as shown in Listing 1) that check whether shadow variables or memory are concrete. In fact, when no symbolic value is involved, there is no need for such instrumentation and the program can be analyzed in concrete mode. In concrete mode, no symbolic emulation will be added to the original code, except for the load and store instructions that may potentially access symbolic values from the shadow memory. Therefore, only the memory access instructions are instrumented to monitor whether the source is symbolic, if so, the execution should switch to the symbolic mode. Section §3.1 introduces how SymFit efficiently performs this check for load and store instructions. For memory stores, the shadow memory of the destination operand will be cleaned since the source operand is always concrete.

Mode Switch. While concrete mode is efficient, instructions that need symbolic emulation should be instrumented to ensure correct symbolic state propagation and constraint collection. Also, SymFit should transition back to concrete mode as soon as symbolic emulation is not necessary. The key challenge is to decide when and how the transition between the two modes should occur. To ensure the proper propagation of symbolic states, the execution should switch to the symbolic mode immediately upon loading symbolic data from the shadow memory. To minimize the instrumentation overhead, the execution should switch to the concrete mode whenever no symbolic values are involved. Moreover, the transition between the two modes should be seamless without disrupting the program's execution. SymQEMU [23] is built on top of the dynamic binary translator QEMU (user mode) [7]. It

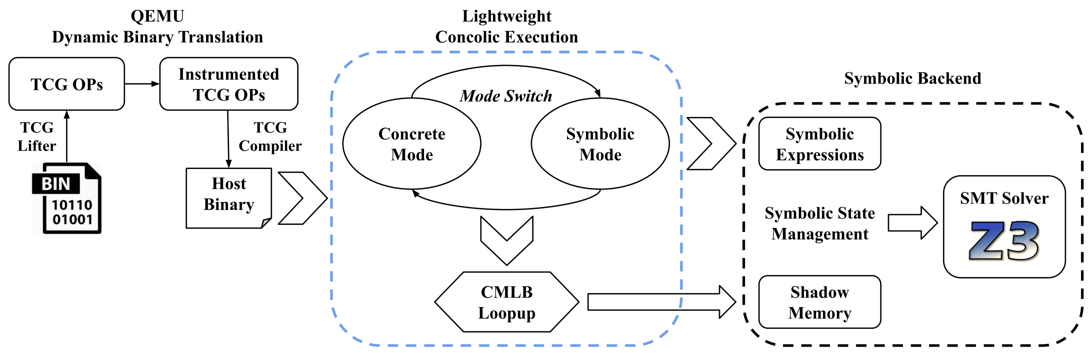

Figure 2: Overview of SymFit

translates and executes the target binary at the basic block level. Before executing a basic block, it first checks whether this block has been translated before, if so, it directly fetches the generated machine code from the code cache and if not, it translates this block of code and stores the generated code into the code cache. To avoid the repeated translation overhead, we designed two code caches for two execution modes correspondingly such that each mode only fetches from or stores to its own code cache.

Transition from Concrete Mode. In the concrete mode, all vCPU registers are concrete, and only the shadow memory may contain symbolic values. Therefore, SymFit only instruments memory loads and stores to detect if a symbolic value is loaded from memory or overwritten by a concrete value. If so, SymFit will automatically switch to the symbolic mode. This switch is triggered by a custom exception: `EXCP_SYM_SWITCH` defined in QEMU's vCPU data structure. When a load from symbolic shadow memory occurs, the exception is raised, forcing the execution to jump back to the main execution loop of QEMU. In the execution loop, this exception is captured and handled by changing the mode flag that instructs how QEMU should instrument the target binary and select the code cache.

Transition from Symbolic Mode. In the symbolic mode, SymFit inserts the full symbolic execution logic into the program. We use the vCPU register states as an indicator to switch from the symbolic mode to the concrete mode. Right before QEMU executes the target basic block in symbolic mode, SymFit checks if all vCPU registers are concrete. If so, no symbolic value is involved in this basic block, and SymFit can safely switch back to the concrete mode for better efficiency.

3.1 Efficient Shadow Memory Checking

SymFit adopts a lightweight concrete execution by only monitoring memory accesses (i.e., load and store). As presented in Table 2, the concrete execution mode results in an average of 3× speedup over SymQEMU on SPEC-CINT benchmarks. However, shadow memory checking still has about an 11× slowdown compared to the vanilla QEMU.

Table 2: Speedup of lightweight concrete mode over SymQEMU on SPEC CINT-2006 with all concrete inputs and overhead of mode switch compared to vanilla QEMU. SymFit-M represents SymFit with only the optimization of mode switching between lightweight concrete mode and symbolic mode.

CINT-2006	SymQEMU	SymFit-M	Speedup	Overhead
hmmer	293.39s	107.85s	2.72×	13.46×
libquantum	17035.43s	4155.04s	4.01×	7.66×
bzip2	4625.94s	1109.15s	4.17×	6.16×
mcf	3072.34s	883.54s	3.48×	3.61×
sjeng	29495.09s	6634.13s	4.45×	6.78×
gcc	1749.03s	518.36s	3.38×	7.21×
xalancbmk	68874.56s	63293.54s	1.09×	205.34×
Geo. Mean			3.08×	11.27×

In SymQEMU, load and store instructions are instrumented with helper function calls to first check whether the destination address is concrete. To optimize for the common case of concrete shadow memory access, SymFit maintains a buffer that stores recently accessed concrete pages. It is inspired by how the system-mode QEMU accelerates the address translation. In the system-mode QEMU, each memory access must go through an address translation from its guest virtual address to its host virtual address. A naïve implementation would also involve a helper function call and the complex address translation logic inside the helper call. To speed up this address translation, the system-mode QEMU adopts software TLB (Translation Lookaside Buffer). It checks this TLB for a quick translation and only goes through the page table traversal when there is a TLB miss. Furthermore, it inlines this TLB lookup logic into the generated code block (the code that is finally executed natively on the host machine), to eliminate the overhead of a helper call. Similarly, we maintain a lookaside buffer for recently accessed concrete memory blocks (in

```c
#define CPU_TLB_ENTRY_BITS 3
#define CPU_TLB_DYN_DEFAULT_BITS 10
typedef struct CPUTLBDescFast {
  /* (n_entries - 1) << CPU_TLB_ENTRY_BITS */
  uintptr_t mask;
  /* The array of TLB entries itself. */
  CPUTLBEntry *table;
} CPUTLBDescFast QEMU_ALIGNED(2 * sizeof(void *));

typedef struct CPUTLBEntry {
  target_ulong page_addr;
} CPUTLBEntry;
static void user_tlb_dyn_init(CPUArchState *env) {
  for (int i = 0; i < NB_MMU_MODES; i++) {
    size_t n_entries = 1 << CPU_TLB_DYN_DEFAULT_BITS;
    env_tlb(env)->f[i].mask =
          (n_entries - 1) << CPU_TLB_ENTRY_BITS;
    env_tlb(env)->f[i].table =
          g_new(CPUTLBEntry, n_entries);
  }
}
```

Listing 2: The design and configuration of the Concrete Memory Lookaside Buffer.

which all memory bytes are concrete). We refer to this buffer as Concrete Memory Lookaside Buffer (CMLB). The CMLB lookup logic is inlined in the native code to avoid the helper call overhead. In this way, since most memory blocks are concrete, we expect the majority of memory accesses will result in CMLB hits and no further actions are needed. Occasionally, if a memory access results in a CMLB miss, meaning at least one byte in that block is symbolic, SymFit goes through the slow path by making a helper call as in SymQEMU.

We follow the design of QEMU's software TLB and reuse some of the TLB data structures for the Concrete Memory Lookaside Buffer. As shown in Listing 2, the buffer is initialized with 1024 (1 << `CPU_TLB_DYN_DEFAULT_BTIS`) entries and is associated with the `CPUArchState` data structure per vCPU. Each entry contains an 8-byte address of a concrete memory block. For load and store instructions in concrete mode, SymFit first looks up the CMLB to check if the target address is concrete. If so, it can safely continue in the concrete mode. If not, a helper function is called to read the shadow memory. Note that a store in the concrete mode will write the destination shadow memory as concrete. For load instructions in symbolic mode, if CMLB hits, the shadow variable of the destination operand will be marked as concrete, and the function call to read shadow memory is skipped. For store instructions in the symbolic mode, SymFit only goes to the fast path when CMLB hits and the source variable is also concrete. The entries in CMLB will be filled with a new block if it is concrete, or evicted if the cached concrete blocks become symbolic.

The granularity of the cached memory blocks in CMLB impacts both the hit rate and overall performance. The original TLB design in QEMU holds the starting address of a 4096-byte page in each entry. However, this level of granularity is inefficient for buffering concrete memory blocks as a small amount of symbolic data would mark the whole page as symbolic. To address this, we evaluated various granularity levels ranging from 32-byte to 2048-byte and selected 512-byte which demonstrated the best hit rate in our evaluation. More details can be found in §5.5.

As presented in Table 3, lightweight concrete mode and shadow memory access (dubbed SymFit-MC) together can achieve an average speedup of over 13 times compared to SymQEMU. The overhead compared to the vanilla QEMU is only 2.65 times.

Table 3: Speedup of the optimizations for concrete instructions and shadow memory access over SymQEMU on SPEC CINT-2006 with all concrete inputs and overhead of mode switch compared to vanilla QEMU. SymFit-MC represents the combination of mode switch and shadow memory access.

CINT-2006	SymQEMU	SymFit-MC	Speedup	Overhead
hmmer	293.39s	19.09s	15.37×	2.38×
libquantum	17,035.43s	907.43s	18.77×	1.67×
bzip2	4,625.94s	354.58s	13.05×	1.97×
mcf	3,072.34s	648.82s	4.74×	2.65×
sjeng	29,495.09s	3,299.05s	8.94×	3.37×
gcc	1,749.03s	269.31s	6.49×	3.75×
xalancbmk	68,874.56s	1,074.76s	64.08×	3.49×
Geo. Mean			13.10×	2.65×

3.2 Symbolic State Management

SymQEMU utilizes the same backend with QSYM [31] to manage symbolic expressions and shadow memory. Recently, SymSan [11] proposed a novel approach that extends the dynamic data-flow analysis framework, DFSan [30], to optimize the management of symbolic expressions and shadow memory. Unfortunately, SymSan requires source code, so it cannot be used directly with binary concolic executors. In this subsection, we explain how we apply the ideas of SymSan to enhance existing binary concolic executors.

Symbolic Expression Management. State-of-the-art concolic executors, such as SymCC [24], SymQEMU [23] and QSYM [31], utilize Abstract Syntax Trees (ASTs) to represent symbolic expressions and their dependencies. When a new symbolic expression is created, an AST node is allocated on the heap to store the new symbolic expression populated based on the source operand(s) and instruction. At a symbolic branch, QSYM looks up the AST tree and unfolds the dependencies to build a Z3 expression for path negation. According to the performance profiling in SymSan, QSYM spends 3% of execution time on AST node allocation and 28% on AST node tracking. To reduce the overhead of allocating and tracking symbolic expressions, SymSan modifies the taint labels in DFSan [30] to represent symbolic expressions and preserves a large, consecutive address space for forward allocation of new labels. As a result, tracking symbolic expressions is a simple look-up of the label array (constant time), and allocating new labels is done by performing an `atomic_fetch_add` to update the last allocated index. Interested readers can refer to SymSan [11] for more technical details. To adopt the design of SymSan, we add helper functions at TCG IR level that interact with SymSan's backend for symbolic state management.

Shadow Memory Management. For variables stored in memory, SYMQEMU uses QSYM to model the shadow memory to store their expressions. In QSYM, the shadow memory mapping is maintained in a red-black tree (`std::map`). To calculate the shadow address for a given application address, the page-level application address is first used as an index to retrieve the target shadow page, then the page offset is added to it to get the shadow address. As a result, the mapping complexity is *O(log(n))*.

```
// Calculate shadow address in QSYM.
std::map<uintptr_t, SymExpr *> g_shadow_pages;
static SymExpr *getShadow(uintptr_t address) {
  shadowPageIt =
          g_shadow_pages.find(pageStart(address));
  if (shadowPageIt != g_shadow_pages.end())
    return shadowPageIt->second + pageOffset(address);

  return nullptr;
}
// SymSan direct shadow memory mapping.
void *shadow_for(uptr addr) {
  return (addr & ShadowMask()) << 2;
}
```

Listing 3: The shadow memory design in QSYM and SymSan

As designed in SymSan [11] and other dynamic taint analysis (DTA) tools [30], direct mapping is the most time-efficient way to maintain shadow memory, which offers constant time (*O(1)*) lookup. SYMFIT employs the direct mapping from SymSan's symbolic backend. Listing 3 shows the shadow memory address translation in SymSan. Despite the fast shadow memory mapping, the shadow memory access still suffers from poor memory access locality. As presented in Table 4, the application memory and shadow memory exhibit a significant distance between them. Therefore, CMLB is still useful in terms of reducing shadow memory access latency.

In conclusion, we improve the performance in symbolic mode by adopting the faster symbolic backend from the source code-based concolic executor SymSan to SYMFIT.

4 Implementation

SYMFIT utilizes the user-mode emulator QEMU [7] to build the front-end and follows the design of SymSan to build the symbolic backend. In this section, we reveal some implementation details of SYMFIT.

TCG Instrumentation. Fundamentally, SYMFIT utilizes the Tiny Code Generator (TCG) of QEMU to perform the instrumentation. TCG first lifts basic blocks of the target binary to an intermediate representation called TCG ops, then compiles TCG ops to machine code that runs on the target architecture. During the dynamic binary translation, additional TCG ops are emitted to invoke library functions from the symbolic execution backend. For each TCG op, the symbolic handler function is added to build corresponding symbolic expressions. Unlike source code-based instrumentation that benefits from compiler optimizations, TCG instrumentation has a limited view of instructions and renders advanced static analysis infeasible. As a result, it is difficult for SYMQEMU to statically determine if a variable is concrete, and remove the inserted symbolic execution logic. SYMQEMU settled for a solution that performs concreteness checks at the cost of library function calls. In SYMFIT, we use vCPU registers to indicate whether TCG variables are concrete in the target basic block. Specifically, in symbolic mode, we add one helper function per basic block to check the states of all registers. If they turn out to be concrete, we can safely avoid the cost of concreteness checks for every instruction in this block.

Symbolic State Management. SYMFIT uses runtime library functions from SymSan's symbolic backend to construct symbolic expressions and send solving queries to the SMT solver. At the core of SymSan, a special form of dynamic data-flow analysis is performed to collect taint labels of program variables and their dependencies. When encountering a symbolic (i.e., tainted) branch, a symbolic expression is reconstructed based on collected taint labels. While SymSan works on LLVM IRs, we instrument TCG IRs to achieve the equivalent functionalities.

SYMFIT supports symbolic data from input files and `stdin`. To achieve this, we hook the syscall wrappers in QEMU (e.g., open, openat, read, and lseek) and assign corresponding labels to the buffer that receives the read bytes. Symbolic data from network interfaces is not supported yet but can be easily extended.

Symbolic Address. SYMFIT uses the same strategy to handle symbolic address as SYMQEMU. Specifically, for a symbolic address, new test inputs are generated to reach other possible addresses. A symbolic address is also associated with the concrete value of the address to ensure correctness.

Memory Layout. SymSan uses direct shadow memory mapping and forward allocation of symbolic expressions. As a result, heap memory regions are preserved in advance to program execution, and a specific memory layout is enforced. To benefit from this design, SYMFIT follows the same memory layout and reserves designated memory regions when QEMU starts. To make sure QEMU-related data structures (i.e., vCPU states) are mapped into the shadow memory, we also modify QEMU's customized memory allocator to allocate these data structures at preserved regions. Listing 3 depicts the memory layout.

Table 4: Memory layout designed for SymSan's backend

Start	End	Desription
0x700000050000	0x800000000000	application memory
0x700000040000	0x700000050000	QEMU's object
0x400010000000	0x700000020000	AST array
0x400000000000	0x400010000000	hash table
0x000000020000	0x400000000000	shadow memory
0x000000000000	0x000000010000	reserved by kernel

Hybrid Fuzzer. In the end-to-end hybrid fuzzing experiment, we reuse the same hybrid fuzzer from SYMQEMU and SymCC [24]. Specifically, the concolic executor takes seeds from fuzzer's seed queue as input and generates new seeds for fuzzer to synchronize periodically. It also maintains a global coverage bitmap for branch filtering. For a fair comparison, we implemented the same branch filters as in SYMQEMU.

5 Evaluation

In this section, we evaluate the performance of SYMFIT, a prototype implementing our proposed optimization schemes atop SYMQEMU. We demonstrate the effectiveness of SYMFIT by answering the following research questions:

- **RQ1: Efficiency.** We investigate the extent to which SYMFIT improves efficiency compared to SYMQEMU under different settings. Additionally, we analyze the individual contributions of each design choice to the overall efficiency improvement.
- **RQ2: Effectiveness.** We evaluate whether SYMFIT can achieve the same level of effectiveness as SYMQEMU in terms of generating new test cases and achieving increased code coverage. In other words, we want to ensure that the efficiency improvement does not come at the cost of compromising the quality of generated test cases and the growth of code coverage.
- **RQ3: End-to-end Hybrid Fuzzing.** We evaluate the contribution of SYMFIT to the end-to-end hybrid fuzzing with respect to code coverage growth and bug-finding performance.
- **RQ4: Security Applications.** We assess the contribution of SYMFIT's efficient symbolic tracing capability on one security application, crash seed deduplication.

5.1 Evaluation Plan

To answer the aforementioned research questions, we evaluate the following configurations:

- **SYMQEMU.** The original SYMQEMU with QSYM backend obtained from the public repository [6].
- **SYMFIT-M.** SYMFIT (QSYM backend) with only mode switch enabled.
- **SYMFIT-MC.** SYMFIT (QSYM backend) with both mode switch and CMLB enabled.
- **SYMFIT-MS.** SYMFIT with only mode switch enabled and SymSan backend.
- **SYMFIT.** Full-fledged SYMFIT with the mode switch, CMLB and new symbolic backend from SymSan.

These configurations allow us to discern the enhancements brought about by each design choice, both individually and collectively. For instance:

- The comparison between SYMFIT-M and SYMQEMU illustrates the gains from the lightweight concrete mode.
- Comparing SYMFIT-MC with SYMFIT-M reveals the advancements due to efficient shadow memory access via CMLB.
- By comparing SYMFIT-MS and SYMFIT-MC, we discern the improvements from the fast symbolic backend.
- Ultimately, comparing SYMFIT and SYMQEMU provides insight into the overall benefits of our concolic executor.

Dataset. We evaluate SYMFIT on several benchmarks, including standard benchmarks (SPEC CINT-2006 [5] and Linux/Unix nbench [3]), and real-world programs (Google's Fuzzbench [2] and unibench, the real-world benchmark presented in UniFuzz [20]). Since QEMU cannot run new versions of SPEC CINT benchmarks due to unsupported instructions, we only selected a few benchmarks from SPEC CINT-2006 as motivating results presented in §2.3. In the evaluation, we use nbench to show the concrete mode overhead of SYMFIT and the baseline concolic executor SYMQEMU.

To answer **RQ1**, we conduct experiments on the standard benchmark nbench to evaluate the speedup of SYMFIT over SYMQEMU in two settings: 1) pure concrete execution and 2) concolic execution without solving. Additionally, we evaluate SYMFIT's performance on real-world programs from Fuzzbench to demonstrate the improvements on practical applications. For **RQ2**, we compare the total execution time and basic block coverage achieved by SYMQEMU and SYMFIT on Fuzzbench programs, using the same input seeds. To answer **RQ3**, we pair SYMQEMU and SYMFIT each with two AFL-2.56 instances, one main instance (in -M mode) and one secondary instance (in -S mode) and evaluate the 24h coverage gain and bug detection capability on unibench programs. For **RQ4**, we use symbolic constraints to cluster PoCs that share the same constraints and evaluate the efficiency in symbolic constraint collection of SYMFIT and SYMQEMU.

Experiment Setup. All evaluations were conducted on a workstation with 96-core Intel Xeon Platinum 8260 processors.

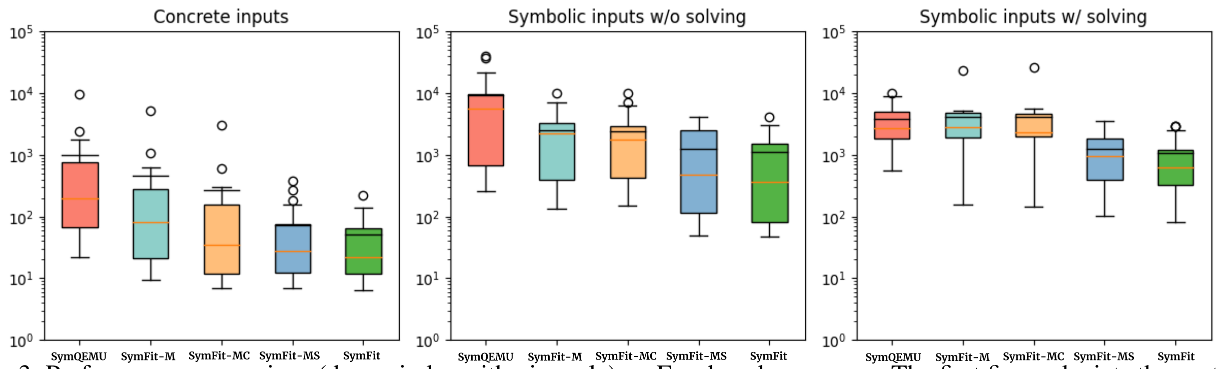

Figure 3: Performance comparison (drawn in logarithmic scale) on Fuzzbench programs. The first figure depicts the runtime of executions with concrete inputs, the middle shows the runtime of executions with symbolic inputs but no constraint solving. The last figure shows the runtime with symbolic input and constraint solving enabled. More statistic details can be found in Appendix Table 8 and Table 9.

The workstation has 1.45T memory with Ubuntu 18.04 operating system and kernel 5.4.0. To ensure a fair comparison, we run each trial in a docker container with one dedicated core assigned. All the evaluation results are averaged across ten experimental trials to reduce the impact caused by randomness from hardware and constraint solver.

Input Selection. To generate test cases for Fuzzbench programs, we used AFL++ to fuzz the target programs for 24 hours and obtained the generated seeds. To avoid bias toward repetitively executed code paths, we used the utility CMIN from AFL++ to prune the seed corpus. For unibench programs, we used the publicly available seed corpus from [20] for better reproducibility.

5.2 RQ1: Efficiency

To evaluate the performance gain in the concrete mode, we run nbench without any symbolic inputs. Consequently, the symbolic backend is not invoked and the only runtime overhead of SymQEMU comes from instrumentation for concrete code blocks, such as helper function calls that check the concreteness of shadow variables and shadow memory. As shown in Table 5, this overhead results in a significant 22× slowdown on SymQEMU compared to the vanilla QEMU. Such overhead in SymFit can be optimized by the lightweight concrete mode and shadow memory access. We evaluate these two strategies individually and show performance improvement from each optimization.

Speedup of Lightweight Concrete Execution. In the first experiment, we evaluated SymFit to understand the performance improvement achieved through the lightweight concrete mode. As all inputs were concrete, all basic blocks were executed in concrete mode. Table 5 presents the results. Compared to vanilla QEMU, SymQEMU exhibited a 21.6× slowdown on the memory index, a 16× slowdown on the integer index, and a 6× slowdown on the floating-point index.

Table 5: Performance of concrete execution on NBENCH

Iterations/sec.	Native	QEMU	SymQEMU	SymFit-M	SymFit-MC	SymFit
NUMERIC SORT	2335.8	1164.3	109.03	325.71	431.41	945.25
STRING SORT	2593.1	301.14	10.421	30.264	124.55	108.67
BITFIELD	1.18E+09	4.86E+08	2.47E+07	9.96E+07	2.21E+08	3.57E+08
FP EMULATION	1244.9	445.14	19.715	95.25	254.45	327.62
FOURIER	2.44E+05	9026	1844.9	5572.2	5895.4	4701.3
ASSIGNMENT	83.345	33.779	1.9065	9.7031	7.2105	23.704
IDEA	19530	5895.5	326.28	2736	2868.7	5308.6
HUFFMAN	12014	2585.4	171.55	867.87	1214.8	2146.7
NEURAL NET	273.27	9.7072	1.151	3.6732	6.7238	6.4019
LU DECOMP	5651.6	322.57	60.067	144.05	258.77	291.33
Score Index						
MEMORY INDEX	85.437	22.962	1.063	4.152	7.861	13.103
INTEGER INDEX	71.122	23.33	1.457	7.25	10.944	19.073
FP INDEX	182.557	7.699	1.272	3.624	5.492	5.212

This slowdown was primarily attributed to the instrumentation required for symbolic emulation in SymQEMU, even when handling concrete inputs. In contrast, SymFit was able to avoid much of the overhead caused by symbolic emulation since all inputs were concrete. The primary overhead in SymFit arose from memory load/store monitoring and shadow register checking at the basic block level. As presented in Table 5, SymFit-M is 3.9× faster on the memory index, 4.97× faster on the integer index, and 2.84× faster on the floating-point index.

Speedup of Efficient Shadow Memory Access. In this experiment, we evaluated SymFit to understand the performance improvement achieved through efficient shadow memory access. In this configuration, recently accessed concrete pages are stored in the Concrete Memory Lookaside Buffer (CMLB), and the cache lookup logic is inlined into the generated code, allowing it to run directly on the host machine. As presented in Table 5, SymFit-MC shows faster execution time than SymFit-M. Compared to SymQEMU, SymFit-MC is 7.4× faster on the memory index, 7.5× faster on the integer index, and 4.7× faster on the floating-point index. We also observed that SymFit shows better performance (1.5×) than SymFit-MC alone. This is attributed to the direct shadow memory mapping mechanism that accelerates the shadow memory lookup when the CMLB misses.

Real-world Applications. We then extended the pure concrete execution evaluation to real-world applications. In this evaluation, we ran Fuzzbench programs with the same seed corpus obtained from 24h fuzzing. Results are presented in Figure 3. Compared to SymQEMU, SymFit-M is 3.65× faster, SymFit-MC is 4.68× faster and SymFit is 8.86× faster.

Concolic Execution without Solving. In this experiment, we evaluated the performance of concolic executors without solving. Compared to pure concrete execution, the overhead in this setting comes from concreteness checking of shadow variables and memory and symbolic state management. According to the result presented in Figure 3, SymFit-M, SymFit-MC, SymFit-MS and SymFit are 3.26×, 3.86×, 9.73× and 12.03× faster than SymQEMU respectively.

Overall, these results demonstrate that SymFit, incorporating both lightweight concrete mode and efficient shadow memory access, outperforms SymQEMU on standard benchmarks and real-world scenarios.

5.3 RQ2: Effectiveness

In this experiment, we enabled constraint solving for each concolic executor. For each target program, we used the coverage bitmap collected from afl for branch filtering to avoid flipping every encountered symbolic branch and placed a 100-second timeout for each execution. Otherwise, the experiment cannot be completed within a reasonable time. The execution times for inputs that did not reach the timeout are shown in Figure 3. Additionally, the basic block coverage was measured using SanitizerCoverage [4] to verify the correctness of constraint solving.

Table 9 shows the coverage achieved by SymQEMU and SymFit. We noticed that SymQEMU reached the timeout on more inputs than SymFit. This observation suggests that SymFit can explore more basic blocks, resulting in higher coverage on certain programs (e.g., freetype + 17.09%, woff2 +17.31%) when provided with the same seed corpus. Overall, the results indicate that SymFit improves the efficiency of concolic execution without compromising correctness.

Based on the results depicted in Figure 3, when solving is enabled, SymFit can still achieve a 6.67× performance speedup over the baseline SymQEMU. Furthermore, even without SymSan's symbolic backend, SymFit-M and SymFit-MC independently achieve an approximate 2× performance speedup, showcasing the contribution of lightweight mode switch and shadow memory access. With the integration of the new backend from SymSan, the performance significantly boosts up to 6.67×, as symbolic state management proves to be heavyweight in SymQEMU's symbolic backend.

5.4 RQ3: End-to-end Hybrid Fuzzing

In this experiment, we pair SymQEMU and SymFit each with two AFL-2.56 instances and evaluate coverage growth and bug detection efficiency of the hybrid fuzzers on unibench, a real-world program benchmark. To ensure a fair comparison, we run each fuzzer/concolic executor in a docker container with one physical CPU-core assigned. The results are obtained by averaging 10 repetitions of 24h fuzzing campaigns to reduce the randomness.

5.4.1 Coverage Efficiency

The coverage growth over time is shown in Figure 4. SymFit achieved faster coverage growth on seven applications (wav2swf, pdftotext, infotocap, mp42aac, objdump, tcpdump, and nm-new) and showed similar performance with SymQEMU on the rest programs. This result indicates that a faster concolic executor can indeed expedite the exploration process during hybrid fuzzing. By achieving faster coverage growth, SymFit is able to explore more paths and execute a broader range of code segments within the same amount of time, potentially leading to the discovery of new vulnerabilities or bugs earlier. However, it is essential to note that in a hybrid fuzzing setting, the fuzzer operates with much higher throughput and actively drives the path exploration process. As a result, the speedup achieved by SymFit is less obvious than the improvement observed in the pure concolic execution evaluations.

5.4.2 Bug Detection Efficiency

To further understand the benefit of having a faster concolic executor, we evaluated the bug detection efficiency of SymFit-HF and SymQEMU-HF. At the end of each hybrid fuzzing run, the crashes are triaged into unique bugs using the same method implemented in unibench: we use ASan [1] to produce the stack trace for each crash and then use three stack frames to de-duplicate the bugs. In this experiment, we compare the time it took to trigger mutual bugs found by both SymFit and SymQEMU. We only counted the first time a bug was triggered. Among evaluated unifuzz benchmarks, seven programs produced crashes but no bug ID was assigned by unibench. There were four programs where both SymFit and SymQEMU did not generate any crash. The results for the rest six programs where mutual bugs were found are presented in Figure 5. As shown, for the same group of mutual bugs found by SymFit and SymQEMU, SymFit spent less time to trigger these bugs. This outcome indicates that SymFit demonstrates improved efficiency in detecting bugs during hybrid fuzzing. By being faster in its execution, SymFit can discover bugs with reduced time-to-trigger, making it an effective tool for detecting vulnerabilities or crashes in real-world applications.

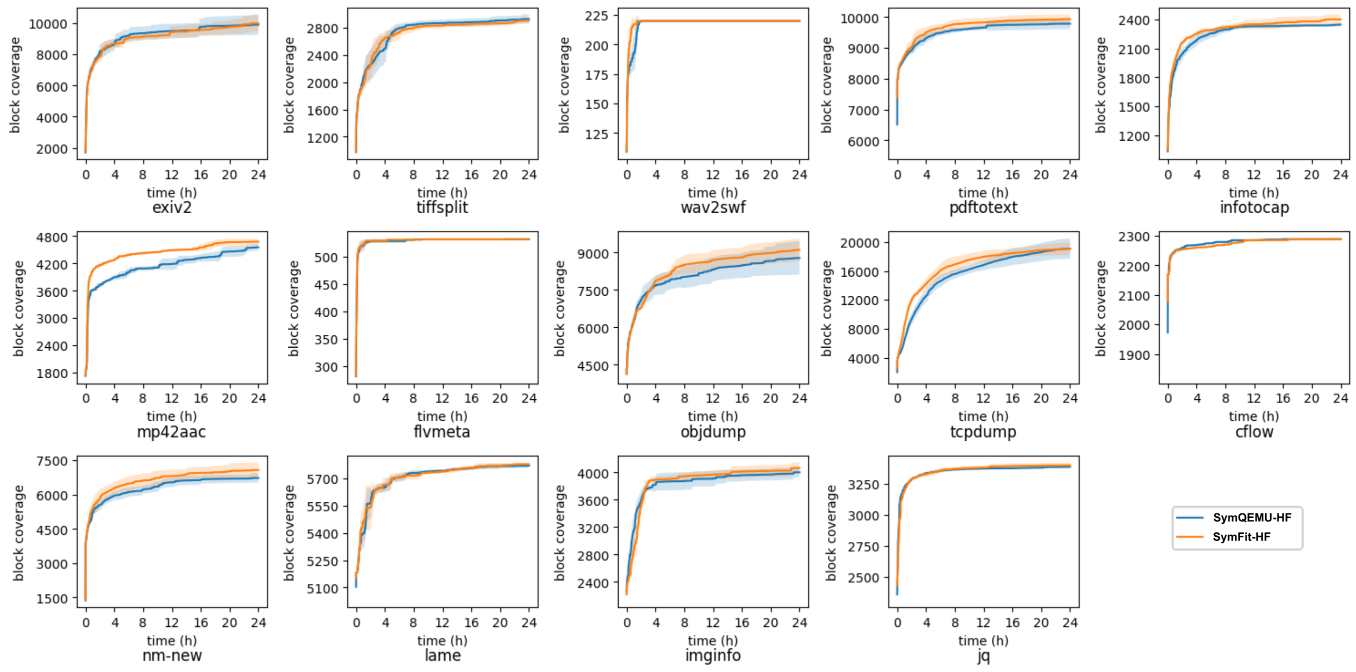

Figure 4: Hybrid Fuzzing Coverage Growth in 24h

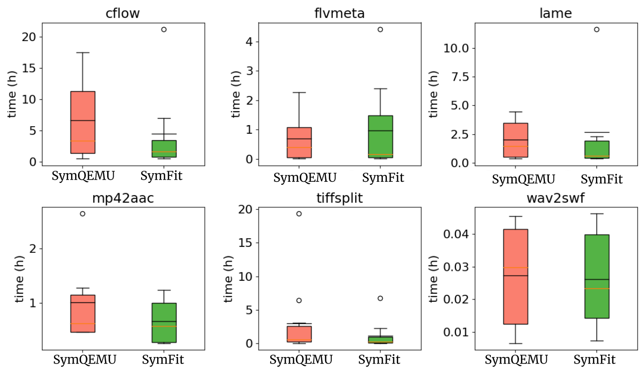

Figure 5: Mutual Bug Time-to-Trigger(h) Result

5.5 Ablation Study

In §5.2, we have assessed the improvement brought by the primary components in SYMFIT, namely the mode switch, Concrete Memory Lookaside Buffer (CMLB), and the integration of the SymSan backend. Here we conducted ablation study to quantify the contribution of each component.

SymSan Backend. Compared to the QSYM backend used in SYMQEMU, SymSan is more efficient in symbolic state management. Specifically, SymSan allocates labels for symbolic variables and preserves a large, consecutive address space for forward allocation of new labels. As a result, allocating new labels is done by performing an `atomic_fetch_add` to update the last allocated index, and label look-up when constructing symbolic expressions occurs in constant time. For variables stored in memory, SymSan utilizes a direct mapping of application memory to maintain the shadow memory, enabling constant time (*O(1)*) shadow memory lookup. In contrast, SYMQEMU employs a red-black tree (`std::map`) for shadow memory mapping, which incurs a logarithmic complexity *O(log(n))*. The performance gains from integrating SymSan's backend are evident by comparing SYMFIT-M (mode switch only) and SYMFIT-MS (mode switch plus SymSan backend), with a 3x speedup, as shown in Table 8.

Mode Switch. SYMFIT features a lightweight execution mode for concrete code blocks and switches between lightweight concrete mode and symbolic mode to minimize unnecessary overhead. As evidenced in Table 6, symbolic blocks constitute only a small percentage of the total execution blocks (9.49% on average) for Fuzzbench benchmarks, demonstrating the significance of maintaining minimal overhead in concrete execution. In the evaluation, we have shown that this optimization alone can bring a 3.26x speedup, as indicated in Table 8 under the SYMFIT-M (mode switch only) configuration.

CMLB. The Concrete Memory Lookaside Buffer (CMLB) is designed to optimize the latency of shadow memory access by buffering recently accessed concrete pages. In this section, we conducted an ablation analysis to measure the CMLB hit rate and explore different levels of granularity. The granularity refers to the size of concrete memory stored in a CMLB entry. The original TLB design in QEMU holds the starting address of a 4096-byte page in each entry. However, this level of granularity is inefficient for buffering concrete shadow memory blocks, as even a small amount of symbolic data within a page would mark the entire page as symbolic. To address this, we experimented with finer granularity levels and compared the hit rates across different granularity, the results of which are presented in Table 6.

Overall, CMLB can achieve a high hit rate, indicating that the cache is frequently accessed instead of the shadow memory. It is important to note that, due to the presence of symbolic memory, the hit rate is lower than the typical TLB hit rate (around 99%), as accessing symbolic shadow memory consistently results in a CMLB miss. As presented in Table 6, the hit rate varied across different levels of granularity and there is no one-size-fit-all configuration. Therefore, we selected a granularity of 512 for our evaluation, as it yielded the highest average hit rate. As presented in Table 8 under the SymFit-MC configuration, the CMLB brings an additional 20% improvement on top of the mode switch.

Table 6: Ablation study for mode switch and CMLB. % of sym blocks represents the percentage of symbolic basic blocks during execution. CMLB hit rate is measured with different granularity, ranging from 32 to 2048. The configuration with the best hit rate is marked as bold.

Fuzzbench	% of sym blocks	CMLB Hit Rate						
		32	64	128	256	512	1024	2048
harfbuzz	8.2	91.2	94.3	96.2	95.4	96.6	96.3	**96.7**
lcms	7.8	92.6	**93.6**	92.2	91.4	91.8	65.3	64.9
libpng	5.3	92.7	94.0	**95.0**	94.0	95.0	94.2	94.3
nm	16.5	87.8	92.0	93.2	**93.7**	92.3	91.4	90.9
openssl	13.7	93.5	**94.3**	93.3	92.2	91.4	90.7	80.4
proj	5.5	93.6	95.7	96.7	96.5	**97.6**	97.2	97.1
woff2	0.5	93.7	95.3	96.4	96.6	98.3	98.6	**99.0**
freetype	6.7	92.9	94.4	95.0	94.7	**96.2**	95.8	95.3
json	6.8	93.3	95.1	96.3	96.4	97.9	98.1	**98.2**
libjpeg	10.0	93.6	95.2	96.2	96.3	98.2	98.5	98.9
objdump	4.5	86.1	90.4	91.8	92.3	**92.9**	92.0	91.4
openthread	2.0	94.9	96.3	97.1	97.1	**97.3**	92.9	93.7
re2	8.3	93.4	**95.0**	95.0	93.4	92.9	88.6	85.8
size	33.3	86.1	90.7	92.1	**92.8**	92.6	89.1	88.6
vorbis	13.0	93.4	95.0	95.9	96.3	98.0	98.5	98.9
xml	9.7	92.0	**93.4**	93.4	91.6	91.1	89.2	86.9
Average	9.4	91.9	94.0	94.7	94.4	**95.1**	92.2	91.3

6 Case Study: Crash Deduplication

At its core, SymFit stands as an efficient symbolic constraint tracer for binary code. In this section, we demonstrate the benefit of SymFit's efficiency in one security application, crash deduplication.

Fuzzers typically output a large set of proof-of-concept (PoC) test cases. These raw findings and crash dumps are often directly submitted to maintainers. Large fuzzing farms, such as ClusterFuzz and OSS-Fuzz, produce a large amount of crash seeds, which exacerbate this problem. An efficient and automated approach is needed to filter out redundant and duplicated crash test cases and cluster them by the root cause. Given a large number of crash seeds produced by fuzzers, a symbolic tracer can examine these crash seeds, and collect symbolic constraints at the last branch before the crash site that is controlled by certain input bytes. This symbolic constraint signature can be used to cluster crash seeds and filter out duplicates and redundancies. The intuition is that the PoCs that trigger identical bugs are likely to follow identical execute paths, therefore sharing the same path constraints.

The above security application highlights the need for efficiency. In this case, the timeout strategy of SymQEMU is no longer valid: without the timeout, SymQEMU could take hours or days to process crash seeds from large fuzzing farms or respond to new exploits, highlighting the need for a more efficient tool.

To trace symbolic constraints for crash deduplication, it is imperative to disable certain features within SymQEMU. namely the timeout strategy and the global branch filter. SymQEMU sets a 90-second timeout on each execution during hybrid fuzzing and adopts a global branch filter to avoid collecting and solving constraints for "uninteresting" branches across executions [23, 24]. While these measures enhance the efficiency in hybrid fuzzing by trading off completeness, crash deduplication demands a more thorough and efficient approach. It requires the tracking of a Proof of Concept (PoC) execution up to the crash site, along with the tracing of all necessary constraints associated with each unique PoC.

To demonstrate the efficacy and efficiency of SymFit in crash deduplication, we evaluated SymQEMU and SymFit on the Magma dataset [18], which contains PoCs and ground truth for grouping PoCs. Specifically, we run each PoC and collect symbolic constraints at the last symbolic branch before the crashing site and back-tracing nested constraints that share the same input dependency, i.e., all precedent branches whose input bytes overlap with the last branch. Then we normalize the variable names in constraints since Z3 might use different names to represent sub-constraints across executions. After this, we compare and group together PoCs that have the same normalized symbolic constraints. Then we measured the efficiency by comparing the total runtime and efficacy by comparing the F1 score [29] of deduplication results, following the calculation in Igor [19]. For the ground truth of PoC grouping, we utilize the ground truth data from the magma dataset to identify and categorize PoCs that reach or trigger identical bugs as belonging to the same group. We excluded benchmarks with no crash seeds provided. As presented in Table 7, the number of PoCs is greatly reduced after deduplication, with an average of 80% F1 score. Note that we used a relatively simply method by grouping identical symbolic constraints. While there are rooms for improving the accuracy of crash analysis, SymFit shows significant better runtime performance as an efficient symbolic tracer; the constraint collection time is greatly reduced from 19 hours to 2.5 hours on average, compared to SYMQEMU.

Table 7: Results of crash seed analysis on Magma benchmarks. The column labeled "after dedup." shows the number of PoC clusters grouped using symbolic constraints produced by SYMFIT

Magma	# of PoC	After dedup.	F1 (%)	Execution Time (h)	
				SYMQEMU	SYMFIT
libpng	634	14	98	33.88	2.48
libtiff	311	15	73	13.77	1.16
libxml2	792	51	80	3.08	0.73
sqlite3	1730	90	69	26.63	5.80
Average	866	42	80	19.34	2.54

7 Discussion

Comparison of Symbolic Backends. In SYMFIT, we ported the symbolic backend from SymSan and compared it with SymQEMU with QSYM backend. While these two backends are similar in path constraint collection and solving, as they can achieve similar coverage when given the same seed corpus, we observed some differences in branch and basic block filtering. As presented in QSYM [31], basic block pruning is employed to reduce constraints that are repeatedly generated from the same code paths, while SymSan is not shipped with such strategies. Therefore, in the hybrid fuzzing experiments, we observed less coverage growth for some programs.

Overhead of Concolic Executors. In general, the overhead of concolic executors comes from instrumentation, symbolic state management, and constraint solving. In this work, we have brought down the overhead of instrumentation and symbolic state management to near-optimal. Nevertheless, we still observed a significant overhead of constraint solving and dynamic binary translation itself, which overshadow our improvement to some extent. The dynamic binary translator, QEMU, translates the target binary at runtime, and the translated code blocks are cached for a single execution. We believe that in a hybrid fuzzing setting, the performance of dynamic binary translation can be improved by sharing the translation cache among every execution. Moreover, with the recent progress in improving the performance of constraint solving [8, 12, 21, 22], more efficient and scalable concolic execution engines can be achieved.

8 Related Work

Compilation-based Concolic Execution. Recent advances in compilation-based concolic execution (e.g., SymCC [24], SymSan [11]) have been shown to improve the performance of concolic execution significantly. When source code is available, compilation-based instrumentation can benefit from compiler optimizations and high-level semantic information (e.g., type) from the source code. While such approaches are sufficient for testing software with source code available, many real-world scenarios require binary-only testing methods, such as firmware, COTS software, shared libraries, etc. SYMFIT focuses on advancing the state-of-the-art binary concolic executor.

Hybrid Instrumentation for Concolic Execution. SymFusion [14] proposed a hybrid instrumentation strategy to minimize the analysis overhead. In particular, this hybrid instrumentation allows users to perform compilation-based instrumentation on selected components of the application and performs dynamic binary instrumentation for the rest of the application at execution time. As such, it can benefit from the high efficiency of compiler-generated code. However, unlike SYMFIT which automatically determines the instrumentation strategies, SymFusion users have to manually decide which components need what instrumentation strategies.

Alternating Execution Modes. To mitigate performance overhead, some previous works proposed to alternate the execution between heavy-weight and light-weight modes. For example, Firm-AFL [32] boots up the IoT firmware in whole-system emulation and ensure the program to be fuzzed runs properly, then it switches to user-level emulation to gain fast execution speed. Only on rare occasions, the execution is migrated back to the system-mode emulation to ensure the correctness of execution. DECAF++ [15] adopts a similar strategy to speed up whole-system taint tracking. It alternates between a lightweight check mode with minimal instrumentation overhead and a track mode with full taint propagation capability. To avoid memory exhaustion, Mayhem [10] introduces hybrid symbolic execution to actively manage memory usage without constantly re-executing the same instructions. When the system reaches a memory cap, it switches to offline execution mode and produces checkpoints to start new online execution tasks later on.

9 Conclusion

In this work, we propose an efficient concolic execution framework with optimizations for the common case, concrete execution. The evaluation showed that SYMFIT can achieve much better performance compared to state-of-the-art binary concolic executor, SYMQEMU. The fast execution speed translates to faster coverage growth and improved efficiency in security applications such as bug detection and crash deduplication.

Availability

The source code of SYMFIT can be found at https://github.com/bitsecurerlab/symfit.git.

Acknowledgement

We would like to thank the anonymous reviewers for their suggestions and comments and Chengyu Song for technical support and discussion on SymSan integration. This work was supported by NSF under award No. 2133487. Any opinions, findings, and 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] AddressSanitizer. https://github.com/google/sanitizers/wiki/AddressSanitizer.

[2] Fuzzbench: Fuzzer benchmarking as a service. https://google.github.io/fuzzbench/.

[3] Linux/Unix nbench. https://www.math.utah.edu/~mayer/linux/bmark.html.

[4] Sanitizer coverage. ttps://clang.llvm.org/docs/SanitizerCoverage.html,2017.

[5] SPEC CPU 2006. https://www.spec.org/cpu2006/.

[6] SymQEMU Github repo. https://github.com/eurecom-s3/symqemu.

[7] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, page 41, USA, 2005. USENIX Association.

[8] Luca Borzacchiello, Emilio Coppa, and Camil Demetrescu. Fuzzing symbolic expressions. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 711–722, 2021.

[9] Cristian Cadar, Daniel Dunbar, and Dawson R Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. 2008.

[10] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. Unleashing mayhem on binary code. USA, 2012. IEEE Computer Society.

[11] Ju Chen, WookHyun Han, Mingjun Yin, Haochen Zeng, Chengyu Song, Byoungyoung Lee, Heng Yin, and Insik Shin. SymSan: Time and space efficient concolic execution via dynamic data-flow analysis. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 2531–2548, August 2022.

[12] Ju Chen, Jinghan Wang, Chengyu Song, and Heng Yin. Jigsaw: Efficient and scalable path constraints fuzzing. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 18–35, 2022.

[13] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. S2E: a platform for in-vivo multi-path analysis of software systems. ACM, 2011.

[14] Emilio Coppa, Heng Yin, and Camil Demetrescu. SymFusion: Hybrid Instrumentation for Concolic Execution. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, ASE '22, 2022.

[15] Ali Davanian, Zhenxiao Qi, Yu Qu, and Heng Yin. DECAF++: Elastic whole-system dynamic taint analysis. In *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)*, pages 31–45, Chaoyang District, Beijing, September 2019. USENIX Association.

[16] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: directed automated random testing. 2005.

[17] Fabio Gritti, Lorenzo Fontana, Eric Gustafson, Fabio Pagani, Andrea Continella, Christopher Kruegel, and Giovanni Vigna. Symbion: Interleaving symbolic with concrete execution. In *2020 IEEE Conference on Communications and Network Security (CNS)*, pages 1–10, 2020.

[18] Ahmad Hazimeh, Adrian Herrera, and Mathias Payer. Magma: A ground-truth fuzzing benchmark. *Proc. ACM Meas. Anal. Comput. Syst.*, 4(3), December 2020.

[19] Zhiyuan Jiang, Xiyue Jiang, Ahmad Hazimeh, Chaojing Tang, Chao Zhang, and Mathias Payer. Igor: Crash deduplication through root-cause clustering. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, CCS '21, page 3318–3336, New York, NY, USA, 2021. Association for Computing Machinery.

[20] Yuwei Li, Shouling Ji, Yuan Chen, Sizhuang Liang, Wei-Han Lee, Yueyao Chen, Chenyang Lyu, Chunming Wu, Raheem Beyah, Peng Cheng, et al. Unifuzz: A holistic and pragmatic metrics-driven platform for evaluating fuzzers. In *USENIX Security Symposium*, pages 2777–2794, 2021.

[21] Daniel Liew, Cristian Cadar, Alastair F. Donaldson, and J. Ryan Stinnett. Just fuzz it: Solving floating-point constraints using coverage-guided fuzzing. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2019, page 521–532, 2019.

[22] Awanish Pandey, Phani Raj Goutham Kotcharlakota, and Subhajit Roy. Deferred concretization in symbolic execution via fuzzing. In *Proceedings of the 28th ACM*

SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2019, page 228–238, 2019.

[23] Sebastian Poeplau and Aurélien Francillon. Symqemu: Compilation-based symbolic execution for binaries. In *ISOC Network and Distributed System Security Symposium (NDSS)*, April 2021.

[24] Sebastian Poeplau and Aurélien Francillon. Symbolic execution with SymCC: Don't interpret, compile! In *29th USENIX Security Symposium (USENIX Security 20)*, pages 181–198, August 2020.

[25] N. A. Quynh and D. H. Vu. Unicorn – the ultimate cpu emulator. `https://www.unicorn-engine.org/`.

[26] Koushik Sen, Darko Marinov, and Gul Agha. Cute: a concolic unit testing engine for c. 2005.

[27] Yan Shoshitaishvili, Ruoyu Wang, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. Firmalice - automatic detection of authentication bypass vulnerabilities in binary firmware. In *22nd Annual Network and Distributed System Security Symposium, NDSS*, 2015.

[28] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *IEEE Symposium on Security and Privacy*, 2016.

[29] Michael S. Steinbach, George Karypis, and Vipin Kumar. A comparison of document clustering techniques. 2000.

[30] the Clang team. Dataflowsanitizer design document. `https://clang.llvm.org/docs/DataFlowSanitizerDesign.html`, 2018.

[31] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. QSYM: A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing. In *Proceedings of the 27th USENIX Security Symposium (Security)*, August 2018.

[32] Yaowen Zheng, Ali Davanian, Heng Yin, Chengyu Song, Hongsong Zhu, and Limin Sun. Firm-afl: High-throughput greybox fuzzing of iot firmware via augmented process emulation. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1099–1114, August 2019.

A Performance of different configurations compared to SymQEMU

Table 8: Performance of concolic execution with no solving on Fuzzbench programs

Program	SymQEMU	SymFit-M		SymFit-MC		SymFit-MS		SymFit	
	Runtime(s)	Runtime(s)	Speedup	Runtime(s)	Speedup	Runtime(s)	Speedup	Runtime(s)	Speedup
harfbuzz	6,372.86	3,253.49	1.96×	2,629.90	2.42×	1,435.19	4.44×	1,387.93	4.59×
lcms	1,022.25	520.93	1.96×	501.45	2.03×	190.09	5.38×	192.37	5.31×
libpng	602.48	200.59	3.00×	175.20	3.43×	65.25	9.23×	47.80	12.60×
nm	1,028.32	392.12	2.62×	410.44	2.50×	71.83	14.32×	74.07	13.88×
openssl	37,566.63	9,901.44	3.79×	8,465.30	4.43×	3,092.26	12.15×	3,098.97	12.12×
proj4	542.10	367.15	1.48×	343.99	1.57×	132.62	4.09×	144.13	3.76×
readelf	1,431.20	772.63	1.85×	693.18	2.06×	271.56	5.27×	54.90	26.07×
woff2	5,617.75	3,041.58	1.85×	2,123.02	2.65×	2,468.08	2.28×	1,534.60	3.66×
freetype	39,538.13	6,139.90	6.44×	5,682.19	6.96×	2,739.58	14.43×	2,764.08	14.30×
jsoncpp	668.10	428.83	1.56×	372.02	1.79×	117.30	5.70×	115.71	5.77×
libjpeg	9,800.92	3,354.55	2.92×	2,511.20	3.90×	3,779.22	2.59×	2,768.12	3.54×
objdump	9,724.00	2,789.92	3.49×	2,699.47	3.60×	1,416.24	6.87×	1,470.67	6.61×
openthread	254.40	135.98	1.87×	139.71	1.82×	80.81	3.15×	82.47	3.08×
re2	8,257.88	6,992.17	1.18×	3,556.12	2.32×	4,200.55	1.97×	4,217.99	1.96×
size	669.30	195.13	3.43×	186.39	3.59×	48.50	13.80×	48.65	13.76×
vorbis	21,629.50	2,229.50	9.70×	1,678.43	12.88×	485.50	44.55×	370.52	58.38×
libxml2	13,982.00	2,202.55	6.35×	1,827.39	7.65×	919.51	15.21×	930.47	15.03×
Geo. Mean			3.26×		3.86×		9.73×		12.03×

Table 9: Execution time of concolic execution with solving (in seconds)

Program	# seeds	Total Execution Time (s)					Speedup	Basic Block Coverage	
		SymQEMU	SymFit-M	SymFit-MC	SymFit-MS	SymFit		SymQEMU	SymFit
harfbuzz	2,955	83,127.10	42,380.61	37,983.57	23,950.27	23,350.93	3.56	9098	**9272**
lcms	157	4,228.63	1,961.08	1,645.23	2,658.01	1,580.09	2.67	2064	1958
libpng	218	9,677.67	4,372.86	4,184.94	1,734.40	1,225.25	7.90	1250	**1256**
nm	249	10,092.71	9,558.61	8,800.40	9,77.07	905.41	11.14	2443	**2850**
openssl	1,577	72,710.73	59,470.24	57,346.02	11,876.38	11,097.04	6.55	12880	**14213**
proj4	770	4,896.43	4,268.11	3,984.75	1,407.56	1,169.63	4.18	3822	**4025**
readelf	604	32,321.24	18,896.14	18,023.36	5,626.86	3,087.03	10.47	5525	**6187**
woff2	548	10,559.96	6,718.48	6,022.52	4,460.72	4,016.44	2.63	3061	**3591**
freetype	4,789	77,083.744	51,294.97	40,423.84	18,964.84	8,141.33	9.47	13368	**15653**
jsoncpp	450	1,967.41	1,509.39	1,363.95	1,027.35	528.99	3.72	1331	**1361**
libjpeg	846	33,397.67	27,725.95	22,479.36	17,370.90	16,920.44	1.97	2674	**2713**
objdump	560	31,472.56	12,140.80	10,819.20	6,118.25	5,812.72	5.41	4000	**4488**
openthread	268	2,966.24	2,431.40	2,195.46	436.86	573.25	5.17	5413	**5509**
re2	1,073	21,701.13	21,350.01	20,075.95	13,653.63	12,882.43	1.68	5060	**5084**
size	207	7,066.35	5,676.14	5,287.81	1,226.68	885.50	7.98	2215	2145
vorbis	526	48,266.28	31,755.67	26,790.59	2,835.14	2,027.14	23.81	1302	**1323**
libxml2	1,952	21,395.87	15,768.22	15,414.94	3,620.96	2,933.85	7.29	7976	7950
Geo. Mean							6.80×		