SYMSAN: Time and Space Efficient Concolic Execution via Dynamic Data-flow Analysis

Ju Chen
UC Riverside

Wookhyun Han
KAIST

Mingjun Yin
UC Riverside

Haochen Zeng
UC Riverside

Chengyu Song
UC Riverside

Byoungyoung Lee
Seoul National University

Heng Yin
UC Riverside

Insik Shin
KAIST

Abstract

Concolic execution is a powerful program analysis technique for systematically exploring execution paths. Compared to random-mutation-based fuzzing, concolic execution is especially good at exploring paths that are guarded by complex and tight branch predicates. The drawback, however, is that concolic execution engines are much slower than native execution. While recent advances in concolic execution have significantly reduced its performance overhead, our analysis shows that state-of-the-art concolic executors overlook the overhead for managing symbolic expressions. Based on the observation that concolic execution can be modeled as a special form of dynamic data-flow analysis, we propose to leverage existing highly-optimized data-flow analysis frameworks to implement concolic executors. To validate this idea, we implemented a prototype SYMSAN based on the data-flow sanitizer of LLVM and evaluated it against the state-of-the-art concolic executors SymCC and SymQEMU with three sets of programs: nbench, the DARPA Cyber Grand Challenge dataset, and real-world applications from Google's Fuzzbench and binutils. The results showed that SYMSAN has a much lower overhead for managing symbolic expressions. The reduced overhead can also lead to faster concolic execution and improved code coverage.

1 Introduction

Concolic execution [8–12, 16, 21, 22, 36, 37, 41, 48] is a powerful program testing tool that can explore code paths effectively without blindly testing inputs that redundantly execute the same code path. For this reason, it has been widely used in finding security vulnerabilities. However, a critical limitation of concolic execution is its scalability. Yun *et al.* [48] reported that KLEE [9] is around 3,000 times slower than native execution, and angr [44] is more than 321,000 times slower. With respect to memory efficiency, we observed that state-of-the-art concolic executors like QSYM [48] and SymCC [36] introduced three orders of magnitude memory consumption blown up.

To understand the key bottlenecks in concolic execution, let us quickly review how a concolic executor works. A concolic executor (CE) maintains two core data structures: a symbolic state $\sigma : v \to \mathtt{sym}_v$ and a set of path constraint PC along the execution path. The symbolic state σ maps each program variable v to its corresponding symbolic expression $\mathtt{sym}_v$. Conceptually, a symbolic expression can be considered as an abstract syntax tree (AST). Consider a simple statement $\mathtt{d} = \mathtt{a} \oplus \mathtt{b}$, where $\oplus$ is an arbitrary binary operator. To update the symbolic state σ, the CE performs the following four operations:

- **Parsing** the instruction $\mathtt{d} = \mathtt{a} \oplus \mathtt{b}$ to extract its operands and operator.
- **Locating** the symbolic expressions $\mathtt{sym}_\mathtt{a}$ and $\mathtt{sym}_\mathtt{b}$ based on $\mathtt{a} \to \mathtt{sym}_\mathtt{a}$ and $b \to \mathtt{sym}_\mathtt{b}$.
- **Creating** a new expression of $\mathtt{sym}_\mathtt{d}$ that has $\mathtt{sym}_\mathtt{a}$ and $\mathtt{sym}_\mathtt{b}$ as child nodes and $\oplus$ as the operator.
- **Updating** σ with the mapping $\mathtt{d} \to \mathtt{sym}_\mathtt{d}$.

One major source of performance overhead for concolic execution is to parse instructions and extract their semantic information. CEs like KLEE [9] and angr [44] interpret each instruction during runtime. As interpretation is usually slower than concrete execution, their performance overhead is extremely high. Recent CEs like QSYM [48], SymCC [36], and SymQEMU [37] replace interpretation with instrumentation (*i.e.*, the parsing is only performed once, either at compile-time, or during instruction translation), thus they are able to significantly reduce the overhead.

Despite these recent successes, the locating, creating, and updating of symbolic expressions still incur significant overhead. For instance, we profiled the state-of-the-art CE SymCC [36] on the program `objdump` and found that at least 65% of its execution time is spent on maintaining allocated AST nodes of symbolic expressions. In addition, the lookup data structure of SymCC, which maintains the mapping between variables and their corresponding symbolic expressions, consumes a 98MB memory footprint when executing one `objdump` instance, whereas native execution consumes only several KBs.

In this work, we aim to further improve concolic execution's scalability by reducing the overhead for maintaining the symbolic state σ. Our key observation is that the concolic execution can be modeled as a special form of *dynamic data-flow analysis* [40]. Therefore, we can significantly reduce the performance and memory overhead for maintaining the symbolic state by leveraging a mature and highly-optimized dynamic data-flow analysis framework.

To verify our hypothesis, we have implemented a concolic executor SYMSAN based on LLVM's data-flow sanitizer (DFSAN). We evaluated SYMSAN with a variety set of programs, including standard benchmarks (Linux/Unix nbench [32]), DARPA Cyber Grand Challenge (CGC) dataset, Google' Fuzzbench [23] test suite, and 4 real-world applications (outside Fuzzbench). The evaluation results showed that compared to state-of-the-art CEs SymCC [36] and SymQEMU [37], SYMSAN achieved 62.0× performance speedup and 6.5× memory footprint reduction. For end-to-end fuzzing, SYMSAN also outperformed SymCC and SymQEMU. In the Google's Fuzzbench benchmark, SYMSAN is 1st by average score and 3rd by average rank. In comparison, SymCC is 6th by average score and 5th by average rank, while SymQEMU is 4th by average and 2nd by average rank.

In summary, this paper makes the following contributions:

- **New design**: we proposed a novel approach to perform concolic execution atop of dynamic data-flow analysis framework and achieved significant performance improvement against state-of-the art concolic execution tools.
- **Open-source**: we open-sourced our prototype implementation SYMSAN[1].
- **Evaluation**: we conducted a comprehensive evaluation to understand the advantages and limitations of our approach.

2 Background and Motivations

In this section, we provide a short review of symbolic/concolic execution, data-flow sanitizer, and our motivations.

2.1 Symbolic Execution

Symbolic execution treats program inputs as symbolic values instead of concrete values. Program variables (including memory and register content) are represented as symbolic expressions, *i.e.*, functions of symbolic inputs. Symbolic execution is a powerful software testing tool because it can cover an execution path using a symbolic input instead of multiple concrete ones.

[1]Our tool is already open-sourced under a different name. SYMSAN is chosen for anonymization.

A symbolic execution engine maintains (i) a symbolic state σ, which maps program variables to their symbolic expressions, and (ii) a set of path constraints *PC*, which is a quantifier-free first-order formula over symbolic expressions [11].

The path constraint *PC* is empty initially. Whenever a conditional statement is encountered, if its predicate is symbolic, the symbolic executor constructs a boolean formula ε (*i.e.*, $\varepsilon =$ `true` for the `if then` branch or $\varepsilon =$ `false` for the `else` branch). The symbolic executor can then check the feasibility of each branch direction by consulting a satisfiability modulo theories (SMT) (*i.e.*, whether $PC \wedge \varepsilon$ and $PC \wedge \neg\varepsilon$ are satisfiable). For each feasible direction, the symbolic executor updates its path constraint *PC* by adding the constraint ($\varepsilon =$ `true` or $\varepsilon =$ `false`) according to the branch direction.

To generate a concrete input that would allow the program to follow the same execution trace, the symbolic execution engine uses *PC* to query an SMT solver for satisfiability and feasible assignments to symbolic values (*i.e.*, input).

Concolic Execution. One disadvantage of classical symbolic execution is that it cannot explore an execution path where a constraint solver cannot solve its path constraints *PC*(*e.g.*, when the constraints contain uninterpreted functions or are too complex). To circumvent the issue, researchers proposed concolic execution (*a.k.a.* dynamic symbolic execution) where symbolic execution is combined with concrete execution. In concolic execution, (i) each variable has two states, one with concrete input and the other with symbolic input, and (ii) the execution path is dictated by the concrete input (*i.e.*, the execution path that is always feasible, regardless of the feasibility of the path constraints). To explore execution paths that deviate from the current concrete path, CE checks the feasibility of the branch target opposite to the concrete direction; if feasible, it generates a corresponding input.

Scalability Issues and Recent Advances. The advantage of symbolic execution over random mutation/generation is the ability to handle complex branch conditions more efficiently (*i.e.*, to find an input that can visit the opposite direction of a branch, solving the corresponding path constraints are faster than fuzzing). The drawback, however, is the lack of scalability. There are three main performance bottlenecks: constraint solving, instruction interpretation, and symbolic state management.

Recently, a line of research work aims to improve the performance of the instruction interpretation. For example, Yun *et al.* [48] observed that KLEE is around 3,000 times slower and angr is more than 321,000 times slower than native execution when testing *md5sum*, *chksum*, and *sha1sum*. They pointed out the slowdown of KLEE and angr is due to their adoption of IR and symbolic emulation, so they proposed a dynamic-instrumentation-based approach directly atop binary instructions.

Based on the observation that collecting symbolic constraints at IR-level is simpler than collecting at the instruction-level [35], Poeplau and Francillon proposed using IR-level instrumentation to (i) avoid symbolic emulation of instructions and (ii) retain the simplicity of symbolic constraints [36]. As a result, their tool SymCC performs significantly faster than both IR-less QSYM [48] and IR-based KLEE [9].

2.2 Dynamic Data-flow Analysis

The dynamic data-flow analysis aims to track the information flow between sources and sinks. Conceptually, a dynamic data-flow analysis framework associates each program variable with a label representing how its value depends on the source. As formalized in [40], a dynamic data-flow analysis is defined by a *policy*, which describes:

- **Label Introduction:** these rules define how labels are introduced into the system.
- **Label Propagation:** these rules define how variables' labels are updated after the execution of an instruction.
- **Label Checking:** these rules define at sinks, what operations to perform.

Dynamic Taint Analysis. When the label is binary (*e.g.*, the label can only be *tainted* or *untainted*), we also call such analysis as dynamic taint analysis (DTA). Because the label is binary, the propagation rules are relatively simple. They can be expressed using propositional logic (*e.g.*, if any of the source operands are tainted, the destination operand is tainted). In DTA, we are mainly concerned about how the execution of a program is affected by taint sources. Two typical applications of DTA are: (i) we mark untrusted inputs controllable by attackers as tainted and check if attackers can control critical data (*e.g.*, the program counter); and (ii) we mark privacy-sensitive data as tainted and check if it will be leaked through the network. As DTA is an instrumental analysis for security applications, many tools have been implemented, which have significantly reduced the cost of the analysis [7, 18, 19, 27, 38].

Forward Symbolic Execution. Forward symbolic execution, as we described in §2.1, can also be modeled as a special form of dynamic data-flow analysis [40]. In concolic execution, labels represent the symbolic expressions of variables. The label source is test input, *e.g.*when the program uses the `read` system call to read the test input file, we mark input bytes as symbolic. The propagation rules define how new symbolic expressions are constructed based on the semantic of each instruction. The label sinks are conditional branches, where we perform two operations (i) update the path constraints and (ii) generate inputs that can visit the opposite branch target.

Data-flow Sanitizer. The data-flow sanitizer (DFSAN) from the LLVM project [46] is a mature and highly-optimized data-flow analysis framework. It performs (LLVM) IR-level instrumentation to insert data-flow tracking logic. Unlike DTA tools optimized to handle binary labels (tainted vs. untainted), DFSAN is designed to track how individual input bytes would affect variables. In other words, a label represents a subset of input bytes affecting the corresponding variable. DFSAN performs set union ($\cup$) instead of using propositional logic ($\vee$) during label propagation.

DFSAN optimizes its performance in several ways. First, it uses the shadow memory implementation from the Address Sanitizer [42] to allow constant-time access to labels corresponding to variables stored in memory. Second, it can introduce shadow variables to store labels corresponding to local variables as DFSAN uses IR-level instrumentation. Third, it performs IR-level optimizations to reduce the access to shadow memory. Finally, it uses an optimized data structure called *union table* to (i) allow fast label access and (ii) reduce the footprint by deduplicating redundant labels (*i.e.*, labels represent the same subset of input bytes).

Besides performance optimizations, DFSAN also provides an interface to implement custom propagation rules for external libraries such as the standard C library.

2.3 Motivation

Despite recent improvements in symbolic execution, the state-of-the-art symbolic executors still impose a significant performance and memory overhead compared to native execution. For example, we tested 24 real-world applications with inputs obtained from 24-hour fuzzing and found that SymCC introduces 8.5x to 32,220x overhead and SymQEMU introduces 226.9x to 39,658.8x overhead than native execution, respectively.

To understand the source of the overhead, we profiled the performance of SymCC and SymQEMU. The result revealed a bottleneck previously overlooked by the existing tools: *the maintenance of the symbolic state* σ*, including representation, storage, and retrieval of symbolic expressions*. Concretely, the existing designs (*e.g.*, the runtime from QSYM [48]) represent a symbolic expression as an on-demand allocated memory object and store those objects in hash map alike data structures. The memory objects are keyed by the variable's address in the application's address space. To ease memory management, some tools adopt smart pointers. As a result, the allocation, store, and retrieval of symbolic expressions introduce non-negligible overhead. Since those operations are the most frequent ones during symbolic execution, their overhead dominates the overall performance of symbolic execution.

In this work, we aim to solve these bottlenecks. Our key observations are (i) forward symbolic execution is a type of dynamic data-flow analysis and (ii) existing dynamic data-flow tools have already spent decades of effort to optimize the allocation, store, and retrieval of labels. Therefore, we can significantly reduce the overhead for maintaining the symbolic state by building a symbolic execution engine on top of a highly-optimized dynamic data-flow analysis framework.

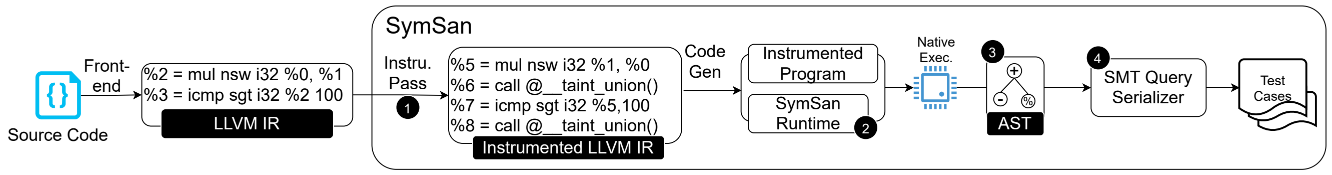

Figure 1: The overall design of SymSan

```
1  ; bool example(int *a, int b) {
2  ;   return (*a) * b > 100;
3  ; }
4  define i1 @example(i32* %0, i32 %1) {
5    %3 = load i32, i32* %0
6    %4 = mul nsw i32 %3, %1
7    %5 = icmp sgt i32 %4, 100
8    ret i1 %5
9  }
10
11 define i1 @"dfs$example"(i32* %0, i32 %1) {
12   ; load taint labels for the arguments
13   %3 =load i32,getelementptr(@__dfsan_arg_tls, 0) ; arg0
14   %4 =load i32,getelementptr(@__dfsan_arg_tls, 1) ; arg1
15   ; load taint label from shadow memory
16   %5 = ptrtoint i32* %0 to i64  ; get shadow_addr(%0)
17   %6 = and i64 %5, -123145302310913
18   %7 = mul i64 %6, 4
19   %8 = inttoptr i64 %7 to i32*
20   %9 = call i32 @__taint_union_load(i32* %8, i64 4)
21   ; load concrete value
22   %10 = load i32, i32* %0
23   ; concrete execution
24   %11 = mul nsw i32 %10, %1
25   ; create a new label to represent (*a) * b
26   %12 = zext i32 %11 to i64  ; extend
27   %13 = zext i32 %1 to i64
28   %14 = call i32 @__taint_union(
29           i32 %9, i32 %4,  ; symbolic operands
30           i16 MUL,   ; operator
31           i8 32,     ; operand size in bits
32           i64 %12, i64 %13 ; concrete operands
33           )
34   ; concrete execution
35   %15 = icmp sgt i32 %11, 100
36   ; create a new label to represent (*a) * b > 100
37   %16 = zext i32 %15 to i64
38   %17 = call i32 @__taint_union(
39           i32 %14, ; symbolic left operand
40           i32 0,   ; zero label for concrete right operand
41           i16 ICMP_LARGER_THAN,  ; operator
42           i8 32,  ; operand size
43           i64 %15, i64 100  ; concrete operands
44           )
45   ; store the label of the return value
46   store i32 %17, @__dfsan_retval_tls
47   ret i1 %15
48 }
```

Figure 2: A running example illustrating how SymSan instrument the target program. Line 1 - 3 shows the source code. Line 4 - 9 shows the uninstrumented LLVM IR compiled from the source code. Line 11 - 48 shows the instrumented LLVM IR.

3 SymSan

In this section, we present the design and implementation details of SymSan.

Insight. Our design goal is to improve the time and space efficiency of the concolic execution. To achieve the goal, we leverage an important insight: *the concolic execution can be viewed as a special form of dynamic data flow analysis*. This observation enables us to build our concolic tool by extending the existing highly-optimized data-flow sanitizer framework. Our design removes two primary bottlenecks in the existing concolic execution tools brought by the management of the symbolic state.

3.1 Overview

Similar to existing instrumentation-based concolic executors like SymCC [36], SymSan performs compile-time instrumentation to insert the logic for introducing, propagating, and checking symbolic expressions. The overall architecture of SymSan is shown in Figure 1. ➊ SymSan takes a compiled LLVM IR as input and instruments the code via a compiler pass. SymSan's run-time ➋ is then linked with the instrumented program to form the final binary. During run-time, the symbolic state (which can be viewed as an abstract syntax forest) ➌ of all program variables is then populated according to the symbolic execution policy. At points of interest (*e.g.*, conditional branches), ➍ SymSan constructs the symbolic formulas of the path constraints, asks an SMT solver to check their feasibility, and generates new test inputs for feasible branch targets.

A Running Example. Figure 2 shows a running example illustrating how SymSan instruments a target program. This program takes two arguments as inputs and returns a boolean. The first argument is an integer pointer (`int *a`), and the second argument is an integer (`int b`). The function first calculates the product of two integers provided by the inputs (`(*a) * b`). Then it compares the product with `100`. If the product is greater than `100`, the function returns `true`; otherwise it returns `false`. We deliberately make the first argument a pointer to show how SymSan accesses shadow memory.

Line 11 - 48 of Figure 2 shows the instrumented version of the function. Recall that given an instruction like `%4 = mul %3, %1`, the core logic of concolic execution consists of three operations:

- **Load**: Locate the symbolic expressions corresponding to `%3` and `%1` from the symbolic state.
- **Creation**: Create a new symbolic expression of that represent the expression `mul %3, %1`.
- **Store**: Bind the new symbolic expression to `%4`.

Next, we describe how these steps are done in SYMSAN.

At Line 14, SYMSAN loads the *label* of `b`, which represents a *unique symbolic expression* (more details in §3.3), from the thread-local storage (TLS). Line 16 - 20 shows how SYMSAN loads the label of `*a`. It first uses the original address (`%0`) to calculate its corresponding shadow address (`%8`) through a fixed mapping scheme (*i.e.*, the shadow address from Address Sanitizer [42]), then directly loads the label from the shadow address. Next, it creates a (new) symbolic expression (`%14`) by passing the two source labels (`%9` and `%4`) and the operator (`MUL`) to the runtime function. Because the product of the inputs (`(*a) * b`) is temporary, its corresponding label (`%14`) will not be permanently stored. Instead, SYMSAN will record, at compile-time, that the label of `%11` is `%14`. Later, when the product is used in the comparison (Line 35), SYMSAN can directly pass `%14` to the runtime function to create the label (`%17`) corresponding to the comparison result (`%15`). Finally, to pass the label of the return value, SYMSAN stores its label in TLS.

In summary, SYMSAN uses labels to represent symbolic expressions, which are stored and retrieved as (i) local (shadow) variables, (ii) thread-local storage, (iii) shadow memory, and (iv) additional arguments (described later). Labels are constructed through a runtime function `_taint_union`. In the next subsection, we explain how SYMSAN optimizes these operations.

3.2 Symbolic State Access

In this subsection, we explain how SYMSAN reduces the performance overhead for storing and retrieving symbolic expressions by comparing it to the closest state-of-the-art tool SymCC [36].

Symbolic Expression Representation. In SymCC, a symbolic expression is either a pointer(usually 64 bits in 64-bit systems) points to a Z3 abstract syntax tree (AST) node (when the simple backend is configured), or points to a QSYM AST node. In SYMSAN, a symbolic expression is a 32-bit label, which is an index to our AST Table (§3.3).

Arguments and Return Value. In SymCC, the symbolic expressions for arguments are passed through a global`std::array`. Similarly, symbolic expressions for return values are passed through a global variable. Consequently, it requires multiple function invocations as well as additional overhead imposed by the C++ container. In addition, this design also limits current SymCC implementation to single-thread programs.

In SYMSAN, labels are passed in two different ways, which are inherited from the DFSAN framework. As shown in Figure 2, the first way is through the per-thread thread-local storage (TLS). Accessing TLS is very fast, and usually only requires a single instruction. For example, on x86, retrieving the label for argument `b` can be done by a single `mov` instruction:

```
; %4 =load i32,getelementptr(@__dfsan_arg_tls, 1)
movq __dfsan_arg_tls@GOTTPOFF(%rip), %rax
```

The second way is to introduce additional arguments. For instance, the wrapper functions for implementing custom symbolic expression constructions for standard C library use additional shadow arguments for each original argument, and a special return label argument:

```
SANITIZER_INTERFACE_ATTRIBUTE size_t
__dfsw_fread(void *ptr, size_t size,
             size_t nmemb, FILE *stream,
             dfsan_label ptr_label,
             dfsan_label size_label,
             dfsan_label nmemb_label,
             dfsan_label stream_label,
             dfsan_label *ret_label)
```

Either way, when symbolic expressions are propagated between functions, SYMSAN is more efficient.

Shadow Memory. For variables stored in memory, most concolic executors use shadow memory to store their labels. To retrieve symbolic expressions from the shadow memory, a CE first needs to convert the original address (a) to its corresponding shadow address. As memory accesses (*i.e.*, `load` and `store`) are very frequent, the speed to perform such translation is critical. In SymCC, shadow memory is implemented in a two-tier mapping. Given an address `addr`, it first uses its page-level address to retrieve the corresponding shadow page through a `std::map`. Once the shadow page is retrieved, the shadow address is calculated by adding the page offset of `addr`:

```
std::map<uintptr_t, SymExpr *> g_shadow_pages;
SymExpr* getShadow(uintptr_t addr) {
  return g_shadow_pages[addr & ~0xfffL]
       + (addr & 0xfffL);
}
```

Due to the lookup through `std::map`, the search complexity is $O(log(n))$.

Because shadow memory is also used by dynamic taint analysis (DTA) tools, they have spent significant efforts to reduce the overhead. So far, the most efficient approach is to use direct mapping, which offers constant time ($O(1)$) lookup. SYMSAN uses the direct mapping shadow memory from the sanitizer family [42]. Specifically, given an address `addr`, its shadow address is calculated as:

```
dfsan_label *shadow_for(uptr addr) {
  return (ptr & ShadowMask()) << 2;
}
```

In summary, SymSan provides much faster shadow memory access.

Shadow Variables. Both SymCC and SymSan use compile-time instrumentation at the LLVM-IR level. Therefore, they enjoy the freedom of introducing additional local variables, which is not feasible for binary-level CEs like QSYM and SymQEMU. Leveraging this advantage, they both use local shadow variables to store symbolic expressions for local variables. Using shadow variables has two main advantages. First, the mapping and lookup are maintained at compile-time, so accessing shadow variables will not introduce additional runtime lookup overhead. Second, it allows compile-time optimizations to remove redundant (stack and shadow memory) accesses. For example, in Figure 2, the product's shadow variable (`%14`) can be directly used to construct the symbolic expression of the return value, without storing and loading from the stack. In summary, both SymCC and SymSan provide optimal access to symbolic expressions of local variables.

Summary. Based on the above analysis, we can see that by leveraging the highly-optimized infrastructure from DFSAN, SymSan can significantly reduce the overhead of storing and retrieving symbolic expressions. Moreover, SymSan uses a more concise representation for symbolic expressions.

3.3 Symbolic Expression Management

In this subsection, we describe how SymSan allocates and stores symbolic expressions in detail. State-of-the-art CEs represent symbolic expressions as memory objects or, more precisely, abstract syntax trees (AST). These tools will *dynamically* allocate a new AST node and populate it based on the source operand(s) to create a new symbolic expression. For example, SymCC [36] offers two different forms of AST. When the simple runtime is configured, SymCC directly uses the AST nodes from Z3. When the QSYM runtime is configured, SymCC uses the AST nodes from QSYM. To ease the memory management, Z3 AST nodes use a reference counter to track living references, while QSYM uses smart pointers `std::shared_ptr` to track living references. Because heap allocation is costly and reference tracking is not free, based on our performance profiling, SymCC spends a considerable amount of time on just allocating (∼3%) and tracking AST nodes (∼28%).

To reduce the overhead of allocating, tracking, and accessing symbolic expressions, SymSan uses an AST table (*i.e.*, an array of AST nodes) to store symbolic expressions. Our observation is that, during dynamic testing (*e.g.*, hybrid fuzzing), because fuzzing throughput has a big impact on the overall fuzzing performance, existing fuzzers all prefer smaller input files [3] and will actively minimize the input files (e.g., `afl-min`). As a result, when processing these small input files, we need to worry too much about memory leaks (e.g., as shown in Figure 5, most concolic execution processes last less than 1 second). Therefore, we organize AST nodes in an array and perform simple forward allocation to allocate new AST nodes.

```
struct dfsan_label_info {
  dfsan_label l1;
  dfsan_label l2;
  u64 op1;
  u64 op2;
  u16 op;
  u16 size;
  u32 hash;
} __attribute__((aligned (8), packed));
```

Figure 3: AST node of SymSan.

AST Nodes. Figure 3 shows the AST node design of SymSan. Each AST node support at most two child nodes (`l1` and `l2`). If a child node is symbolic, its corresponding label will be non-zero, which refers to a subtree. As mentioned above, in SymSan, labels are indices in the AST table (array). If a child node is concrete (*i.e.*, not symbolic), its label will be `0`, and the corresponding concrete value will be stored in the data fields (`op1` and `op2`). `op` stores the operator over the subtree(s). `size` stores the size of operand(s) in bits. `hash` is a hash value of the tree, which is used for deduplication (§3.4). To make it easier to share symbolic expressions, we use the `packed` attribute to prevent the compiler from re-ordering the fields.

AST allocation. SymSan uses a simple forward allocation strategy to allocate new AST nodes. Specifically, SymSan preserves large enough consecutive virtual addresses (see Table 1) for the AST table during initialization. To allocate a new node, it tracks the last label previously allocated (*i.e.*, the largest array index in use) and performs an `atomic_fetch_add` to update the last label. This allows SymSan to allocate a new AST node with a single instruction. The use of `atomic_fetch_add` also allows SymSan to support multi-thread programs.

3.4 Additional Optimizations

Although using simple forward allocation is fast, we can quickly exhaust the fixed size AST table if we blindly allocate new AST nodes every time `_taint_union` is invoked. To address this issue, we designed some optimizations to reduce the size of the consumed AST table entries and improve SymSan's memory efficiency.

Deduplication. The first obvious strategy to reduce the number of allocated AST nodes is deduplication. Before allocating a new AST node, we will check if an identical node already exists. If so, we will reuse the existing one instead of allocating a new node. This is done through a reverse lookup table. In particular, SymSan uses a hash table to map AST nodes back to their labels. Whenever two labels need to be merged, SymSan first queries the hash map to see if it had recorded the corresponding label for the potentially new AST node

$(l_1, l_2, op_1, op_2, op, size)$. If so, it reuses the label returned by the hash map; otherwise, it allocates a new label (AST node).

Because the lookup process involves checking whether two AST nodes are identical and our AST nodes are not small, such comparison could be expensive. Therefore, we need a faster way to check whether two nodes are identical. We use a hash table implementation with chaining to resolve collisions for simplicity. This also requires us to apply a good hash algorithm to avoid frequent collisions. We adopted the Merkle hash tree to meet these requirements. Specifically, each AST node has a hash, which is calculated as follows:

- If the node is a leaf node (*i.e.*, an input byte), its has equal to its label.
- If the node is an intermediate node, its hash is calculated based on its child nodes.
- If a child node is a concrete value, its hash is `0`.

With this hash value calculated for each AST node, when checking if two AST nodes are identical, we will first check if their hash values match; if not, we do not need to check the rest fields. This hash value is also used to access the hash table slot.

Finally, hash table entries are also allocated using a simple forward allocator. To better support multi-thread programs, we also adopted a lock-free implementation.

Load and Store Simplification. In traditional concolic execution, both `load` and `store` operations work at byte granularity. As a result, loading data larger than one byte will involve several *concat* operations; and storing data larger than one byte will result in several *extract* operations.

For example, consider a simple assignment statement with two 32-bit integers: $x = y$, where y is symbolic. When the load operation is recorded at the byte granularity, SymSan needs to create three new AST nodes to concatenate the four individual bytes. To make the matter worse, when storing L_x back to memory, SymSan needs to create an additional four AST nodes to extract individual bytes from the symbolic expression.

In order to increase the label space utilization and simplify the symbolic expressions, SymSan implements additional optimizations for load and store operations. First, SymSan uses a special operator *uload* to express loading a sequence of bytes:

$$label := (uload, l_{start}, size, size)$$

where L_{start} represents the label of the first byte and *size* indicates how many bytes are loaded. When handling a load operation, SymSan will first check if the *uload* operation is applicable (*i.e.*, reading a consecutive of input bytes) before falling back to the *concat* way. Second, when handling store operations, if the label is a result of *uload* operation, SymSan will directly extract labels of the corresponding bytes from the *uload* operation.

3.5 Interactions with External Libraries

Similar to DFSAN, SymSan provides two ways to support external libraries. First, we can instrument the dependent libraries using SymSan, and statically link it with the target program. Most of the Fuzzbench programs we evaluated in §5 follow this way. For libraries that cannot be instrumented, such as `glibc`, we use custom wrappers to implement special label propagation rules. Using a custom wrapper also simplifies the symbolic expressions based on domain knowledge.

Label Introduction. SymSan introduces symbolic labels where test inputs are read. For instance, if the underlying `fread` operation is successful, we will mark bytes in the output buffer as symbolic input bytes, based on their offsets from the beginning of the test input file.

Label Propagation. SymSan also uses custom wrapper functions to implement special propagation rules. Two typical examples are `memcpy` and `memcmp`. In `memcpy`, besides copying the concrete data from the source buffer to the destination buffer, SymSan also needs to propagate labels corresponding to the data in the source buffer to the data in the destination buffer. As `memcmp` is frequently used to check against magic numbers or keywords, we introduced a special higher-order operator *fmemcmp* to symbolize the return value of `memcmp`. Later, if the return value is used in a conditional branch, we can reconstruct the corresponding formula (*e.g.*, bytes in the first buffer must equal the bytes in the second buffer).

4 Implementation

In this section, we reveal some implementation details of our SymSan. SymSan is implemented based on the data-flow sanitizer (DFSAN) [46], which is part of the LLVM compiler toolchain.

Table 1: Memory layout of the program for taint analysis.

Start	End	Description
0x700000040000	0x800000000000	application memory
0x400010000000	0x700000020000	ast table
0x400000000000	0x400010000000	hash table
0x000000020000	0x400000000000	shadow memory
0x000000000000	0x000000010000	reserved by kernel

Memory Layout. SymSan uses directly mapping shadow memory to store labels of program variables stored in memory. Achieving this goal requires 64-bit address space and a special memory layout. Table 1 shows the memory layout of an instrumented program.

To enforce this memory layout, we wrote a linker script to restrict the application memory range, which can avoid colliding with other designated regions. Once the program starts, the runtime library of SymSan reserves the designated regions, so the OS kernel will not allocate virtual addresses within these regions to the application.

Note that although the preserved regions are enormous, the OS kernel will not map physical pages to the addresses until needed.

Label Introduction. To assign labels to input bytes, SymSan instruments file-related functions. In our current prototype, we only support symbolic data from an input file and `stdin`; symbolic data from the network is not supported yet but can be easily extended. When the program opens a file that should be symbolized, SymSan calculates the size of the file and reserves the input label entries. When the program reads from the file, SymSan calculates the offset (within the file) and the size to be read and assigns the corresponding labels to the target buffer that receives the read bytes.

Currently, the following functions are supported: `getc`, `fgetc`, `gets`, `fgets`, `read`, `fread`, `pread`, `getline`, `getdelim`.

Label Propagation. Our label propagation policies are almost identical to DFSAN, the only difference is that when combining two labels, we will construct symbolic expressions. The following (bitvector) operations are supported:

- Bit-wise operations: `bvnot`, `bvand`, `bvor`, `bvxor`, `bvshl`, `bvlshr`, `bvashr`;
- Arithmetic operations: `bvneg`, `bvadd`, `bvsub`, `bvmul`, `bvudiv`, `bvsdiv`, `bvurem`, `bvsrem`;
- Truncation and extension: `bvtrunc`, `bvzext`, `bvsext`;

In our current prototype, we do not support floating point and vector operations, for a fair comparison with SymCC [36] and SymQEMU [37], which also do not support non-integer operations. We also do not support the intrinsic functions of LLVM IR.

Label Checking. In our current prototype, we consider `br` and `switch` instructions as data-flow sinks (*i.e.*, coverage-oriented). For `br` instruction, SymSan checks whether it is conditional; if so, whether its condition is symbolic. For `switch` instruction, SymSan treats each case as a comparison between the condition variable and the case value. For branch targets controlled by symbolic values, SymSan will generate new test inputs for branch target(s) other than the concrete one.

Symbolic Addresses. In our current prototype, we use the same strategy as QSYM [48] and SymCC [36] to handle symbolic addresses. Specifically, SymSan will (1) generate new test inputs to visit other possible addresses; and (2) bind the symbolic address in the current execution trace to its concrete value to ensure correctness.

Nested Branches. One particular challenge when solving path constraints is that solving a single branch predicate alone is insufficient. In our current prototype, we use QSYM's [48] approach to identify nested branches based on data dependencies: finding all precedent branches whose input bytes overlap with the current branch. This strategy is also used by SymCC and SymQEMU.

Supporting run-time libraries. SymSan cannot perform label propagation correctly for code inside an uninstrumented library due to source-code-based instrumentation. For the standard C library, we implemented custom wrapper functions to propagate labels. For the standard C++ library, we instrumented `libc++` from LLVM.

Hybrid Fuzzer. We also implemented a hybrid fuzzer to evaluate SymSan in the end-to-end fuzzing. Overall, our hybrid fuzzer follows the same cross-seeding design as previous hybrid fuzzers [21, 30, 45, 48]. Specifically, SymSan maintains its own FIFO seed queue and a global coverage bitmap. SymSan periodically synchronizes the seeds from the fuzzer's queue. Each time, SymSan fetches a seed in the FIFO queue, executes the seed symbolically and generates new inputs. If the generated inputs cover new code, they are added back to the seed queue. For a fair comparison, we use the same branch filters as SymCC [36], so both tools will roughly flip the same amount of branches.

5 Evaluation

In this section, we evaluate the performance of SymSan to answer the following research questions.

- **RQ1: Time efficiency.** Does SymSan impose less run-time overhead than the state-of-the-art CEs for maintaining the symbolic state? If so, by how much?
- **RQ2: Space efficiency.** Does SymSan use less memory than the state-of-the-art CEs? If so, by how much?
- **RQ3: Effectiveness.** Can test cases generated by SymSan achieve the same or higher code coverage than the state-of-the-art CEs?
- **RQ4: End-to-end fuzzing.** Can SymSan improve the performance of end-to-end hybrid fuzzing?
- **RQ5: Security impacts.** Can SymSan improve the performance of bug finding?

Experimental Setup. All our evaluations were performed on a server with an Intel(R) Xeon(R) E5-2683 v4 @ 2.10GHz (40MB cache) and 512GB of RAM, running Ubuntu 16.04 with Linux 4.4.0 64-bit.

Baseline. We mainly evaluate SymSan against two state-of-the-art CEs: SymCC [36] and SymQEMU [37]. We believe the comparison with SymCC is especially meaningful as both CEs perform compile-time instrumentation at the LLVM IR level, and use Z3 as the constraint solver. For hybrid fuzzing, we include the state-of-the-art fuzzer AFL++ [20] for comparison.

5.1 Dataset

Standard Benchmark. We choose nbench [32] to evaluate the instrumentation overhead of SymSan and baseline CEs.

Table 2: Performance results for pure concrete execution on NBENCH

Tests (Iterations/s)	Native	SYMSAN	SymCC	SymQEMU
NUMERIC SORT	1958	170.26	30.044	15.153
STRING SORT	2425.5	717.2	1.2361	1.2757
BITFIELD	8.76e+08	3.72e+07	1.06e+07	5.34e+06
FP EMULATION	1104	42.952	17.019	4.9317
FOURIER	71699	67760	8958.3	261.54
ASSIGNMENT	99.488	3.6895	0.96664	0.28006
IDEA	17358	572.91	280.11	85.503
HUFFMAN	5969.2	305.26	82.372	32.832
NEURAL NET	158.68	7.428	0.66788	0.17407
LU DECOMPOSITION	3785.3	149.1	26.15	8.4286
Score Index				
Memory Index	80.272	6.218	0.315	0.167
Integer Index	53.834	2.632	0.828	0.298
Floating-point Index	88.594	10.661	1.362	0.184

We did not use SPEC CPU benchmark because its test inputs are too large for evaluated CEs—they all run out of memory.

DARPA Cyber Grand Challenge. CGC programs remove the use of system calls, enabling a fair comparison between source-based and binary-based concolic executions tools and are widely used in the evaluation of state-of-the-art CEs [36, 37, 48]. We follow the same evaluation procedure as previous work, we used PoVs (proofs of vulnerability) as inputs for evaluation. We excluded programs that require inter-process communication and programs on which baseline CEs failed to generate inputs.

Real-world Programs. We evaluated 23 real-world programs shown in Table 5. 16 programs are from Google's Fuzzbench [24], 4 programs are from binutils.

Inputs Selection. To obtain the test inputs for real-world applications, we used AFL++ to fuzz the target programs for 24 hours and obtained the generated seeds as test inputs. To avoid bias toward repetitively executed code paths, we used the utility `cmin` from AFL++ to prune the seed corpus. For bintuils, we used the publicly available seed corpus from [43] for better reproducibility.

5.2 Performance

The performance overhead of an instrumentation-base concolic executor can be classified into the following four categories, which we evaluated separately.

- **Instrumentation.** The overhead from additional code injected to the target program.
- **Symbolic state access.** The overhead for accessing the symbolic expressions correspond to program variables.
- **Symbolic state management.** The overhead for creating and updating symbolic expressions.
- **Constraint solving.** The overhead from consulting an SMT solver.

5.2.1 Pure Concrete Execution

We ran programs in nbench [32] natively (without instrumentation), and with instrumentation of different concolic executors. When running the programs (pinned to a dedicated CPU core) with concrete inputs, the concolic executors will not invoke its symbolic backend. In this way, we can measure the instrumentation overhead of each concolic executor. Table 2 reports the results. Compared to native execution, SYMSAN is 12.9× slower on memory index, 20.5× slower on integer index, and 8.35× slower on the floating-point index. SymCC and SymQEMU are much slower than SYMSAN. SymCC is 254.8× slower on memory index than native execution, 65.0× slower on integer index, and 65.0× slower on the floating-point index. SymQEMU is 480.7×, 180.7×, and 481.5× slower than native execution, respectively. We also noticed that SYMSAN performed much better than SymCC on the memory index. We believe this is due to the direct-mapping-based shadow memory scheme used by SYMSAN.

5.2.2 Pure Taint Propagation

In this experiment, we measure the performance overhead of pure symbolic state accesses. To do so, we disabled the real creation and storage of symbolic expressions; instead, we "simulate" the creation of new symbolic expressions by simply returning a new label for SYMSAN, and a new expression pointer (cast from an increasing integer) for SymCC. This comparison shows the benefit of SYMSAN's shadow memory implementation.

CGC. Following the same procedure as the previous papers [36, 37, 48], we used the first PoV input to test each CGC challenge. We enforced the same 5-minute timeout for each execution as [48] for easier comparison with the numbers reported in previous papers. The results are shown in Figure 4. Compare to native execution, SYMSAN (`SymSan-Taint`) has 1.3 times slowdown and SymCC (`SymCC-Taint`) has 4.9 times slowdown in average execution time. In the metric of median execution time, SYMSAN has 1.8 times slowdown and SymCC has 127.1 times slowdown.

Real-world applications. For real-world applications, we collected the overall running time for every concolic executor executing all seeds. The results are shown in Figure 5. Overall, in the metric of average execution time, SYMSAN (`SymSan-Taint`) introduces 3.7 times overhead compared to the native execution, while SymCC (`SymCC-Taint`) introduces 18 times overhead. In the metric of median execution time, the numbers are 2.25 times and 4.5 times respectively for SYMSAN and SymCC.

Both experiments show that SYMSAN's sanitizer-based shadow memory implementation is much faster than SymCC's.

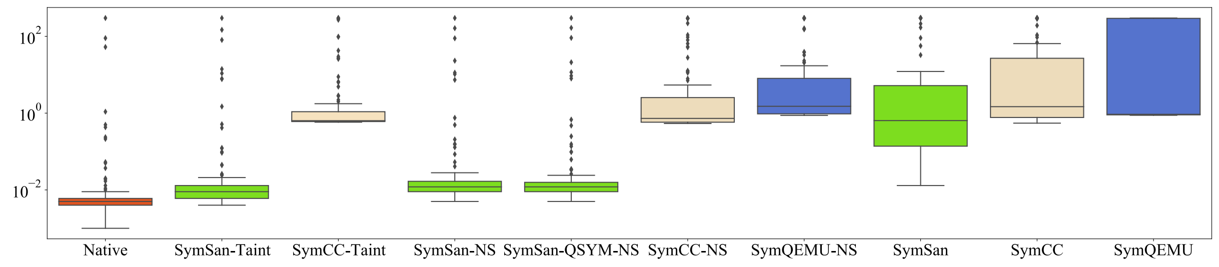

Figure 4: Execution time of 102 CGC challenge binaries, using the first PoV as inputs. The figure is drawn in logarithmic scale. SymSan-Taint and SymCC-Taint measure pure symbolic state access overhead (§5.2.2). SymSan-NS, SymSan-QSYM-NS, SymCC-NS, and SymQEMU-NS measure the concolic execution overhead without solving (§5.2.3). SymSan, SymCC, and SymQEMU measure the full-fledge concolic execution overhead (§5.2.4).

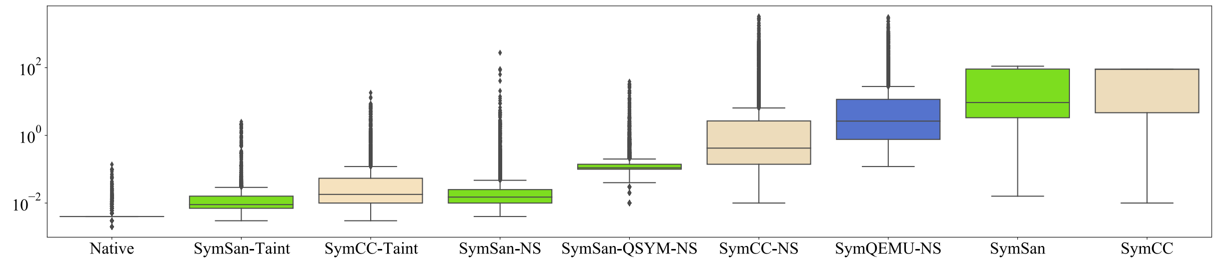

Figure 5: Execution time for the real-world programs. The figure is drawn in logarithmic scale. SymSan-Taint and SymCC-Taint measure pure symbolic state access overhead (§5.2.2). SymSan-NS, SymSan-QSYM-NS, SymCC-NS, and SymQEMU-NS measure the concolic execution overhead without solving (§5.2.3). SymSan, SymCC, and SymQEMU measure the full-fledge concolic execution overhead (§5.2.4).

5.2.3 Concolic Execution without Solving

In this section, we evaluate the performance of concolic executors without solving. Overhead measured in this experiment is an accumulation overhead of instrumentation, symbolic state access, and symbolic state management. To better reflect the benefit of SYMSAN's AST table, we included a configuration SYMSAN-QSYM that uses the same shadow memory implementation to access symbolic expressions, but uses QSYM's backend to manage symbolic expressions (i.e., the same as SymCC and SymQEMU). The ablation study for the two additional optimizations presented in §3.4 is in Appendix. Overall, constraint deduplication improved the performance by 16% and load/store simplification added another 6% speed up on top of deduplication.

CGC. We used the same procedure as described above. The execution time for each program is visualized in Figure 4. In the metric of average execution time, SYMSAN (`SymSan-NS`) is 1.36 times slower than the native execution, and SYMSAN with QSYM backend (`SymSan-QSYM-NS`) is 1.37 times slower. Since each CGC program is only run for 5 minutes with a single input, the performance difference between SYMSAN's AST table and QSYM's backend is not very large. In comparison, SymCC (`SymCC-NS`) and SymQEMU (`SymQEMU-NS`) are 5.3 and 7.2 times slower than SYMSAN respectively. In the metric of median execution time, the numbers are 146 and 301 times respectively for SymCC and SymQEMU.

Real-world applications. The overall running time for every concolic executor executing all seeds is presented in Table 5 (in Appendix). The distribution for inputs that did not timeout is shown in Figure 5. Note that SymQEMU timeout on all inputs so it is not shown. Similarly, 75% of inputs timeout on SymCC, and only 65% of inputs timeout on SYMSAN. Overall, in the metric of average execution time, SYMSAN (`SymSan-NS`) introduces a 17.4× overhead compared to the native execution, while SymCC (`SymCC-NS`) and SymQEMU (`SymQEMU-NS`) introduce 2243× to 5026× overhead respectively. As a result, SYMSAN achieves 128× performance speedup over SymCC and 288× over SymQEMU. In the metric of median execution time, SYMSAN achieves 28× performance speedup over SymCC and 176× over SymQEMU. In this experiment, as each execution trace is much longer than CGC's, SYMSAN's AST table exhibits much better performance than QSYM's backend (7.3× speedup), and only imposes 1.7× overhead over `SymSan-Taint` (in median execution time).

5.2.4 Full-fledged Concolic Execution

We enabled constraint solving for each concolic executor and check if a faster symbolic backend would improve the overall concolic execution speed.

CGC. We used the same setup for CGC as in the previous CGC experiment. We collected the execution time for each program and the result is visualized in Figure 4. As we can see, SYMSAN is still faster than SymCC and SymQEMU, but its advantage becomes smaller. This is because constraint solving takes a significant portion of the overall concolic execution

Table 3: Execution time of concolic execution engines with solving (in seconds). SymCC cannot build *sqlite*3, crashes on 70% of seeds for *libpng*, cannot generate any new input for *proj*4, and has solving crashes on all *re*2 seeds. Coverage is basic-block coverage measured by SanitizerCoverage.

Program	#seeds	Total Execution Time (sec)			Basic Block Coverage		
		SymSan	SymCC	SymQEMU	SymSan	SymCC	SymQEMU
readelf	604	27,712	41,695	49,947	7,660	6,067	3,807
objdump	560	43,612	44,052	47,627	4,789	4,668	4,528
nm	249	6,106	14,127	18,628	3,386	2,746	2,755
size	207	3,517	8,920	15,976	2,448	2,198	2,226
libxml2	1952	2,234	51,588	33,172	8,161	8,014	8,022
proj4	770	696	N/A	7,181	4,500	N/A	4,286
vorbis	526	27,476	44,606	45,531	1,396	1,396	1,396
re2	1073	47,536	N/A	37,596	5,139	N/A	5,136
woff2	548	18,432	28,843	45,693	3,454	3,464	3,460
libpng	218	907	N/A	13,781	1,286	N/A	1,283
libjpeg	846	61,672	56,888	59,159	2,754	2,744	2,744
lcms	157	800	3,278	6,545	2,073	2,047	2,106
freetype	4789	245,684	288,627	338,307	16,171	16,013	15,294
harfbuzz	2955	1,472	144,190	196,759	9,536	9,471	9,351
jsoncpp	450	667	3,733	2,584	968	966	941
openthread	268	1,280	1,474	3,562	5,565	5,565	5,533
openssl	1577	48,180	119,786	134,798	11,887	11,900	11,893
mbedtls	491	3,479	29,569	39,968	4,186	4,145	4,140
sqlite3	5253	111,029	N/A	421,947s	32,843	N/A	36,124
curl	1343	501	1,312	102,180s	13,171	13,122	13,140

time, which was also reported in previous work [36].

Real-world applications. For real-world applications, we placed a 90-second timeout for each execution. Otherwise, the experiments cannot be completed in a reasonable time. Note that placing a timeout for each concolic execution is a common practice adopted by both SymCC and SymQEMU. The results are shown in Table 3. The execution time distribution for inputs that did not timeout is shown in Figure 5. As we can see, with solving enabled, SymSan still enjoys a performance speedup over SymCC and SymQEMU. But again, the advantage is smaller compared to concolic execution without solving.

5.3 Memory Consumption

In this section, we evaluated the memory usage by SymSan and compared it with SymCC, since both are source-based concolic executors. We chose *maximum resident size* as the memory usage metric for each comparison. The visualized result is shown in Figure 6. As we can see, SymSan introduces a much smaller memory overhead than SymCC (3.4× vs. 82.4×). The result also shows that our AST table is more memory efficient than the QSYM backend.

5.4 Code Coverage

In this experiment, we compared SymSan' code coverage with SymCC and SymQEMU on CGC programs and real-world applications.

CGC. For CGC, we measured the coverage by following the method introduced by Yun *et al.* [48]. For each program, we used an AFL coverage map to collectively record the coverage for all generated test cases. For each program, let A be the coverage map for SymSan and B the coverage map for our comparison target (SymCC or SymQEMU). The difference between A and B is then calculated as below as per [48]:

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

The score will be in the range of $[-1.0, 1.0]$, where 1.0 means SymSan not only covers all paths that are covered by other concolic executors but also covers some unique paths. Our results are visualized in Figure 7. As we can see, SymSan has a similar code coverage as SymCC, it covers slightly more than SymCC in 83 programs while covers less in 19 programs.

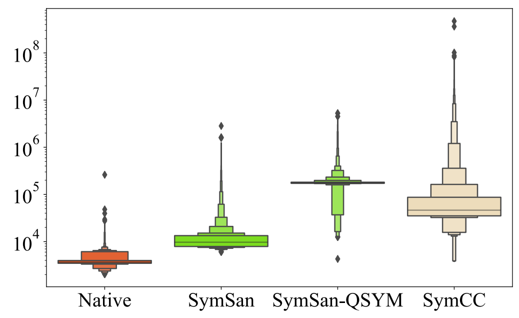

Figure 6: The peak resident size for each concolic execution without solving. The average memory consumption for native execution is 4.1MB. SymSan consumes 14.1MB memory in average, while SymCC consumes about 337.9MB in average.

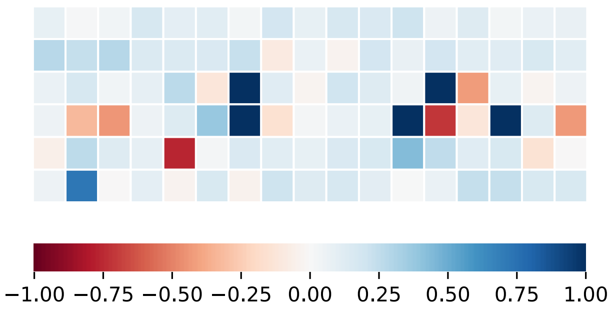

Figure 7: Coverage score comparing SymSan and SymCC per tested program (102 CGC challenge binaries in total). We re-use the visualization method introduced in [48]: Blue colors indicate that SymSan found more paths, red colors indicate that that SymCC found more and white colors indicate equal coverage. A deeper color indicates a larger coverage difference. SymSan performs better on 83 programs and worse on 19 programs.

Real-world Applications. For each real-world application, We measured the basic-block coverage for all generated inputs using SanitizerCoverage [1]. The results are in Table 3. In most programs, SymSan has similar coverage as SymCC and SymQEMU except *sqlite*3, where SymSan covers significantly less than SymQEMU, this is because SymQEMU is a binary-based concolic executor and can handle external libraries better than SYMSAN. In 18 out of 20 programs tested, the code coverage by SYMSAN is more than or equal to SymCC. In the rest of the 2 programs, SYMSAN covers slightly less than SymCC. In 16 out of 20 programs tested, the coverage of SYMSAN is more than or equal to SymQEMU. In the rest of the 4 programs, SYMSAN covers less than SymQEMU.

5.5 Hybrid Fuzzing

In this evaluation, we plugged SYMSAN into the hybrid fuzzing scheme to check if a faster concolic executor helps in the end-to-end fuzzing.

Fuzzbench. We first compared SYMSAN with other popular concolic executors and fuzzers on Google's Fuzzbench dataset [24]. We use AFL++ (commit `70bf4b4` with the default build and fuzz options[2]) for hybrid fuzzing. The experiment is conducted by Google on its cloud. Due to the page limit, we only provide a summary here. The full report can be retrieved at `https://anonymoussubmission2022.github.io`. Out of 12 fuzzers (11 state-of-the-art and 1 from us), SYMSAN is 1st by average score and by average rank, For median coverage, SYMSAN leads in 9 programs, and AFL++ only leads in 3 programs.

We also compared SYMSAN's performance with other concolic executors including SymCC [36], SymQEMU [37], and Fuzzolic [6] based on their publicly available experiment report[3]. The merged report can be retrieved at `https://anonymoussubmission2022.github.io/symsan`. SYMSAN is the first by average score and third by average rank. We summarized the median coverage reached in 24 hours for each concolic executor tool in Table 4. SYMSAN leads in 7 programs, both SymQEMU and Fuzzolic lead in 4 programs, and SymCC leads in 3 programs.

Local fuzzing. For programs that are not included in the Fuzzbench dataset, we conducted hybrid fuzzing in our local environment. For the baseline, we added AFL++ with commit `70bf4b4` and cmplog enabled. SYMSAN, SymCC [36], and QSYM [48] used the same hybrid fuzzing configuration as described in the QSYM's tutorial: a concolic executor paired with two AFL (version 2.56b) instances, one master and one slave. In addition, the concolic executor has a 90-second timeout for executing each seed. For each concolic executor/fuzzer, we executed 10 fuzzing trials, each for 24 hours. To ensure a fair comparison, we use the Fuzzbench's configuration to run each fuzzer/concolic executor in a docker container with 1 physical CPU-core assigned.

[2] `https://github.com/google/fuzzbench/blob/master/fuzzers/aflplusplus/fuzzer.py`

[3] `https://www.fuzzbench.com/reports/experimental/2021-07-03-symbolic/index.html`

Table 4: Comparing SYMSAN with other state-of-the-art symbolic executors based on their publicly available Fuzzbench results. The metric is median coverage reached in 24 hours. We show the results of 15 programs where all tools generate valid results. SYMSAN takes lead in 10 out of 14 programs.

Target	SYMSAN	SymCC	SymQEMU	Fuzzolic
curl	**17926.5**	17622.0	17564.5	17599.5
freetype	**28080.0**	25496.0	24028.0	26371.0
harfbuzz	**8656.0**	8482.5	8482.5	8515.0
lcms	3506.5	3701.5	3656.0	**3770.0**
libjpeg	3802.5	3810.5	**3819.0**	3814.0
libpng	2136.0	1914.5	**2149.5**	2146.5
libxml2	**12799.0**	11097.0	12305.0	12072.0
libxslt	**18799.0**	18577.0	18592.5	18515.0
mbedtls	**8353.5**	8260.0	8244.5	8268.0
openssl	13772.5	**13777.0**	**13777.0**	13767.5
openthread	5833.5	**5935.0**	5862.5	5912.0
proj4	**7262.0**	5365.0	5314.0	5836.5
re2	3516.0	3521.5	3519.0	**3544.5**
vorbis	2166.0	2167.5	**2168.0**	**2168.0**
woff2	1872.0	1934.0	1934.0	**1936.5**

The result is shown in Figure 8. SYMSAN can achieve higher final coverage than other tools on the four programs from binutils. For the rest three programs, SYMSAN performs similarly to other CEs but lags behind AFL++. There are two main reasons. First, the current implementation of SYMSAN only supports tracking symbolic expressions over integers, while AFL++ is type-agnostic. Second, SYMSAN's support for external libraries is limited by its custom wrappers. As a result, certain important label propagation rules could be missing. SYMSAN was lagging behind SymCC on *objdump* at the beginning because SymCC only imports seeds marked with `+cov`, while SYMSAN will execute all imported seeds.

5.6 Security Implications

Recent research has shown a strong correlation between a testing tool's ability to achieve code coverage and its ability to find bugs [4]. Similarly, recent research also showed that leveraging symbolic execution to solve bug triggering constraints can also improve a hybrid fuzzer's ability to find bugs [15]. Based on these observations, we expect that SYMSAN can also help with finding bugs.

In this subsection, we present case studies to demonstrate SYMSAN's ability on finding bugs. Specifically, we used programs with known bugs from the Magma benchmark [26], and evaluated three hybrid fuzzers: (1) `symsan` (SYMSAN with AFL), (2) `symsan_sec`, with inserted security assertions for divide-by-zero (as a simulation to [15]), and (3) `symccafl` (SymCC with AFL). The full results are shown in Table 7 in Appendix. The present numbers are averaged over 10 trials. Following are a few highlights.

- **AAH001 (CVE-2018-13785)** is a divide-by-zero bug in

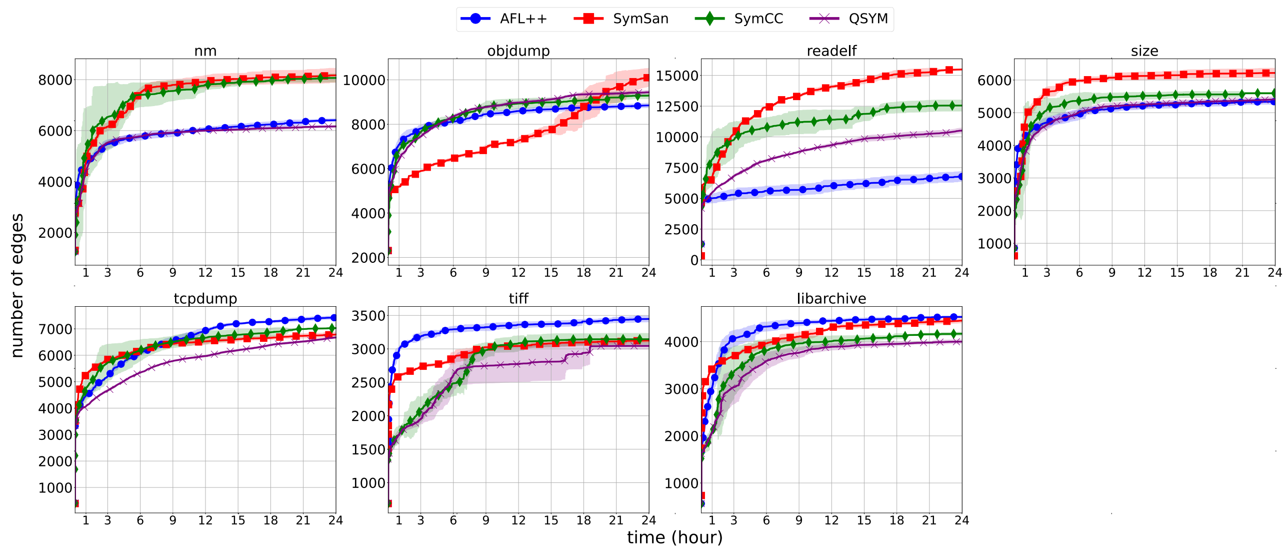

Figure 8: Edge coverage growth over time for local fuzzing.

libpng and can be used to demonstrate the utility of the symbolic executor. symsan can trigger the bug in 8 minutes, while symsan_sec can trigger it much faster—in just 29 seconds. For comparison, symcc takes 57.04 minutes to trigger the bug, and the fastest mutational fuzzer Honggfuzz uses 17.7 hours [26].

- **AAH017 (CVE-2019-7663)** is a NULL-pointer dereference bug in libtiff. symsan_sec is the fastest hybrid fuzzer to trigger the bug, using 5.84 hours. symsan uses 7.69 hours to trigger this bug. In comparison, symccafl takes 11.18 hours to trigger the bug and the fastest mutational fuzzer moptafl takes 5.2 hours [26].
- **AAH055 (CVE-2016-2108)** is an out-of-bound read issue in openssl. Both symsan and symsan_sec can trigger this bug, using 2.53 minutes and 2.51 minutes, respectively. However, it takes 15.68 minutes for symccafl to trigger the bug.

6 Limitations and Future Work

Our current design and implementation of SymSan have some limitations that we plan to address in future work.

Memory Layout. Similar to other sanitizers, SymSan requires a special memory layout (§4) for the shadow memory. So, the current implementation of SymSan requires the target program to be compiled in 64-bit mode and cannot support programs that can only be compiled in 32-bit.

Supported Operations. The current prototype of SymSan does not support floating-point and vector operations. We made this choice because other state-of-the-art concolic executors like QSYM [48], SymCC [36], and SymQEMU [37] also do not support these operations. We plan to add the support in the future.

Constraint Solving and Path Explosion. We believe that by bringing down the overhead of constraint collection in CE to near-optimal, future research can focus on solving other issues of CE, such as constraint solving and path/seed prioritization. Indeed, we have also seen some recent progress in improving constraint solving performance [5, 13, 29, 34]. So ultimately, we, as a community, can achieve efficient and scalable concolic execution.

7 Related work

Besides the works already discussed in §2, the following works are related to this paper.

Concolic Execution. Besides the performance issue, another challenge for concolic execution is the path explosion problem. To mitigate this problem, SAGE [22] proposed utilizing generational search to increase the number of generated test cases in one execution, which has been adopted by most following-up work. Dowser [25] proposed using static analysis to guide concolic execution to places where it is more likely to have buffer overflow vulnerabilities. To compensate the scalability problem of concolic execution engines, another popular approach is to combine concolic executing with fuzzing [31, 45, 48, 49]. In these approaches, path exploration is mostly done by the fuzzer, who is more effective at exploring easy-to-flip branches. Whenever the fuzzer encounters a hard-to-flip branch, it asks the concolic execution engine to solve it.

Taint-guided Fuzzing. Dynamic taint analysis (DTA) [33] is another popular technique to improve the efficiency of fuzzing. TaintScope [47] utilizes DTA to discover and bypass checksum checks and target input bytes that can affect security system library calls. Vuzzer [39] uses DTA to locate magic number checks and then changes the corresponding input bytes to match the magic number. Steelix [28] also uses DTA to bypass magic number checks but has better heuristics. Redqueen [2] uses the observation that input byte could indirectly end up in the program state (memory), so by directly comparing values used in compare instructions, it is possible to infer such input-to-state relationships without expensive taint tracking. Neuzz [43] approximates taint analysis by learning the input-to-branch-coverage mapping using a neural network, which can then predict what inputs bytes can lead to more coverage. Eclipser [17] exploits the observation that many branch predicates are either linear or monotonic with regard to input bytes and solves them using binary search. Angora [14] is another close approach to SymSan. It uses DFSAN to collect input dependencies of conditional branches, then performs gradient-guided search to find inputs that can flip the corresponding branch.

8 Conclusion

In this work, we propose leveraging highly-optimized data-flow analysis framework to reduce the performance and the memory overhead of a concolic executor. Evaluation of our prototype built upon LLVM's data-flow sanitizer validated our idea—our prototype SymSan can significantly outperform the state-of-the-art concolic executors SymCC and SymQEMU.

Acknowledgments

This work is supported, in part, by the National Science Foundation under Grant No. 2046026, No. 2133487, and the Office of Naval Research under Award No. N00014-17-1-2893. 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] Sanitizer coverage. `https://clang.llvm.org/docs/SanitizerCoverage.html`, 2017.

[2] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. Redqueen: Fuzzing with input-to-state correspondence. In *Annual Network and Distributed System Security Symposium (NDSS)*, 2019.

[3] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as markov chain. In *ACM Conference on Computer and Communications Security (CCS)*, 2016.

[4] Marcel Böhme, László Szekeres, and Jonathan Metzman. On the reliability of coverage-based fuzzer benchmarking. In *International Conference on Software Engineering (ICSE)*, 2022.

[5] Luca Borzacchiello, Emilio Coppa, and Camil Demetrescu. Fuzzing symbolic expressions. In *International Conference on Software Engineering (ICSE)*, 2021.

[6] Luca Borzacchiello, Emilio Coppa, and Camil Demetrescu. Fuzzolic: mixing fuzzing and concolic execution. *Computers & Security*, page 102368, 2021.

[7] Erik Bosman, Asia Slowinska, and Herbert Bos. Minemu: The world's fastest taint tracker. In *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2011.

[8] Ella Bounimova, Patrice Godefroid, and David Molnar. Billions and billions of constraints: Whitebox fuzz testing in production. In *International Conference on Software Engineering (ICSE)*, 2013.

[9] Cristian Cadar, Daniel Dunbar, and Dawson R Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2008.

[10] Cristian Cadar, Vijay Ganesh, Peter M Pawlowski, David L Dill, and Dawson R Engler. Exe: automatically generating inputs of death. In *ACM Conference on Computer and Communications Security (CCS)*, 2006.

[11] Cristian Cadar and Koushik Sen. Symbolic execution for software testing: three decades later. *Communications of the ACM*, 56(2):82–90, 2013.

[12] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. Unleashing mayhem on binary code. In *IEEE Symposium on Security and Privacy (Oakland)*, 2012.

[13] Ju Chen, Jinghan Wang, Chengyu Song, and Heng Yin. Jigsaw: Efficient and scalable path constraints fuzzing. In *IEEE Symposium on Security and Privacy (Oakland)*, 2022.

[14] Peng Chen and Hao Chen. Angora: Efficient fuzzing by principled search. In *IEEE Symposium on Security and Privacy (Oakland)*, 2018.

[15] Yaohui Chen, Peng Li, Jun Xu, Shengjian Guo, Rundong Zhou, Yulong Zhang, Tao Wei, and Long Lu. Savior: Towards bug-driven hybrid testing. In *IEEE Symposium on Security and Privacy (Oakland)*, 2020.

[16] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. S2e: A platform for in-vivo multi-path analysis of software systems. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2011.

[17] Jaeseung Choi, Joonun Jang, Choongwoo Han, and Sang Kil Cha. Grey-box concolic testing on binary code. In *International Conference on Software Engineering (ICSE)*, 2019.

[18] James Clause, Wanchun Li, and Alessandro Orso. Dytan: a generic dynamic taint analysis framework. In *International Symposium on Software Testing and Analysis (ISSTA)*, 2007.

[19] Ali Davanian, Zhenxiao Qi, Yu Qu, and Heng Yin. Decaf++: Elastic whole-system dynamic taint analysis. In *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2019.

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

[21] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: directed automated random testing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2005.

[22] Patrice Godefroid, Michael Y Levin, and David A Molnar. Automated whitebox fuzz testing. In *Annual Network and Distributed System Security Symposium (NDSS)*, 2008.

[23] Google. fuzzer-test-suite. `https://github.com/google/fuzzer-test-suite`, 2017.

[24] Google. Fuzzbench: Fuzzer benchmarking as a service. `https://google.github.io/fuzzbench/`, 2020.

[25] Istvan Haller, Asia Slowinska, Matthias Neugschwandtner, and Herbert Bos. Dowser: a guided fuzzer to find buffer overflow vulnerabilities. In *USENIX Security Symposium (Security)*, 2013.

[26] Ahmad Hazimeh, Adrian Herrera, and Mathias Payer. Magma: A ground-truth fuzzing benchmark. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 4(3):1–29, 2020.

[27] Vasileios P Kemerlis, Georgios Portokalidis, Kangkook Jee, and Angelos D Keromytis. libdft: Practical dynamic data flow tracking for commodity systems. 2012.

[28] Yuekang Li, Bihuan Chen, Mahinthan Chandramohan, Shang-Wei Lin, Yang Liu, and Alwen Tiu. Steelix: program-state based binary fuzzing. In *ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, 2017.

[29] Daniel Liew, Cristian Cadar, Alastair F Donaldson, and J Ryan Stinnett. Just fuzz it: solving floating-point constraints using coverage-guided fuzzing. In *ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, 2019.

[30] Kin-Keung Ma, Khoo Yit Phang, Jeffrey S Foster, and Michael Hicks. Directed symbolic execution. In *International Static Analysis Symposium*, 2011.

[31] Rupak Majumdar and Koushik Sen. Hybrid concolic testing. In *International Conference on Software Engineering (ICSE)*, 2007.

[32] Uwe F. Mayer. Byte magazine's bytemark benchmark program. `https://www.math.utah.edu/~mayer/linux/bmark.html`, 2017.

[33] James Newsome and Dawn Xiaodong Song. Dynamic taint analysis for automatic detection, analysis, and signaturegeneration of exploits on commodity software. In *Annual Network and Distributed System Security Symposium (NDSS)*, 2005.

[34] Awanish Pandey, Phani Raj Goutham Kotcharlakota, and Subhajit Roy. Deferred concretization in symbolic execution via fuzzing. In *International Symposium on Software Testing and Analysis (ISSTA)*, 2019.

[35] Sebastian Poeplau and Aurélien Francillon. Systematic comparison of symbolic execution systems: intermediate representation and its generation. In *Annual Computer Security Applications Conference (ACSAC)*, 2019.

[36] Sebastian Poeplau and Aurélien Francillon. Symbolic execution with symcc: Don't interpret, compile! In *USENIX Security Symposium (Security)*, 2020.

[37] Sebastian Poeplau and AurÃ©lien Francillon. SymQEMU: Compilation-based symbolic execution for binaries. In *Annual Network and Distributed System Security Symposium (NDSS)*, 2021.

[38] Feng Qin, Cheng Wang, Zhenmin Li, Ho-seop Kim, Yuanyuan Zhou, and Youfeng Wu. Lift: A low-overhead practical information flow tracking system for detecting security attacks. In *Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2006.

[39] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. Vuzzer: Application-aware evolutionary fuzzing. In *Annual Network and Distributed System Security Symposium (NDSS)*, 2017.

[40] Edward J Schwartz, Thanassis Avgerinos, and David Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *IEEE Symposium on Security and Privacy (Oakland)*, 2010.

[41] Koushik Sen, Darko Marinov, and Gul Agha. Cute: a concolic unit testing engine for c. In *ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, 2005.

[42] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. Addresssanitizer: A fast address sanity checker. In *USENIX Annual Technical Conference (ATC)*, 2012.

[43] Dongdong She, Kexin Pei, Dave Epstein, Junfeng Yang, Baishakhi Ray, and Suman Jana. Neuzz: Efficient fuzzing with neural program learning. In *IEEE Symposium on Security and Privacy (Oakland)*, 2019.

[44] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, and Christopher Kruegel. Sok:(state of) the art of war: Offensive techniques in binary analysis. In *IEEE Symposium on Security and Privacy (Oakland)*, 2016.

[45] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *Annual Network and Distributed System Security Symposium (NDSS)*, 2016.

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

[47] Tielei Wang, Tao Wei, Guofei Gu, and Wei Zou. Taintscope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection. In *IEEE Symposium on Security and Privacy (Oakland)*, 2010.

[48] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. Qsym: A practical concolic execution engine tailored for hybrid fuzzing. In *USENIX Security Symposium (Security)*, 2018.

[49] Lei Zhao, Yue Duan, Heng Yin, and Jifeng Xuan. Send hardest problems my way: Probabilistic path prioritization for hybrid fuzzing. In *Annual Network and Distributed System Security Symposium (NDSS)*, 2019.

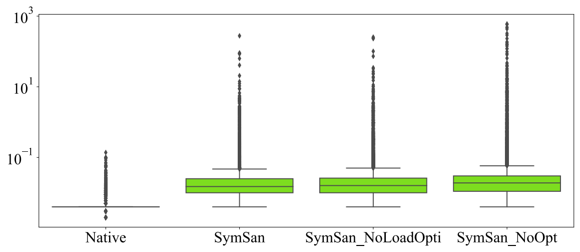

Figure 9: Execution time for the real-world programs. The figure is drawn in logarithmic scale. SymSan-NoOpti is SYMSAN without expressions deduplication and load/store optimization. SymSan-NoLoad is without load/store optimization).

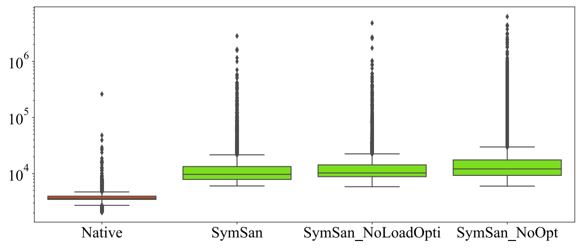

Figure 10: The peak resident size for the real-world programs. The figure is drawn in logarithmic scale. SymSan-NoOpti is SYMSAN without expressions deduplication and load/store optimization. SymSan-NoLoad is without load/store optimization).

Appendix

This section includes additional evaluation results that cannot fit into the main paper.

Concolic Execution without Solving. Table 5 shows the execution time of SYMSAN, SymCC, and SymQEMU to collect symbolic constraints without solving (§5.2.3).

Effectiveness of the Two Additional Optimizations. Figure 10 and Figure 9 shows the execution time and the peak resident size distribution of (1) native execution, (2) full-fledge SYMSAN, (3) SYMSAN without load/store optimization, and (4) SYMSAN *without* expression deduplication *and* load/store optimization. As we can see, both optimization techniques can help reduce the execution time. To find out how often the load/store optimization can be applied, we also measured the ratio of `uload/concat`, the result is `13.8:1`.

Statistics of Collected Constraints. Figure 11 shows (1) the maximum number of tracked expressions (in the number of AST nodes). The median numbers of track expressions are 22,270 and 20,245 for SYMSAN and SymCC respectively.

Table 6 shows the statistics of all the processed constraints in experiments conducted in §5.4 (we considered 16 programs in this statistics, that is all programs in Table 3, but excluded `sqlite, libpng, re2, proj` where SymCC do not generate legitimate results). SYMSAN solves about twice the number of constraints than SymCC.

Bug Finding. Table 7 shows the evaluation results on the Magma benchmark.

Table 5: Execution time of concolic execution engines collecting all constraints without solving (in seconds). SymCC cannot build *sqlite*3. SymCC crashes on 70% of seeds for *libpng*.

Program	#seeds	Native	SymSan		SymCC			SymQEMU		
			Time	vs. Native	Time	vs. Native	vs. SymSan	Time	vs. Native	vs. SymSan
readelf	604	1.6s	10.1s	6.3x	462.8s	289.3x	45.8x	2916.2s	1822.6x	288.7x
objdump	560	2.7s	24.4s	9.0x	2097.0s	776.7x	85.9x	21913.1s	8116.0x	898.1x
nm	249	0.8s	3.3s	4.1x	169.1s	211.4x	51.2x	2598.6s	3248.3x	787.5x
size	207	0.6s	2.2s	3.7x	91.5s	152.5x	41.6x	1520.2s	2533.7x	691.0x
libxml2	1952	6.7s	36.5s	5.4x	2966.5s	442.8x	81.3x	12040.0s	1797.0x	329.9x
proj4	770	2.6s	10.8s	4.2x	22.2s	8.5x	2.1x	776.8s	298.8x	71.9x
vorbis	526	2.6s	267.2s	102.8x	83772.2s	32220.0x	313.5x	103113.2s	39658.9x	385.9x
re2	1073	7.3s	215.2s	29.5x	16655.4s	2281.6x	77.4x	221078.9s	30284.8x	1027.3x
woff2	548	2.2s	295.4s	134.3x	19812.6s	9005.7x	67.1x	12918.0	5871.8x	43.7x
libpng	218	0.7s	2.5s	3.6x	N/A	N/A	N/A	1126.1s	1608.7x	450.4x
libjpeg	846	3.1s	83.0s	26.8x	42243.3s	13626.9x	509.0x	49465.2s	15956.5x	596.0
lcms	157	0.8s	4.9s	6.1x	26.9s	33.6x	5.5x	4335.0s	5418.8x	884.7x
freetype	4789	15.9s	202.1s	12.7x	16139.3s	1015.1x	79.9x	98562.2s	6198.9x	487.7x
harfbuzz	2955	9.4s	22.3s	2.4x	11903.4s	1266.3x	533.8x	16788.0s	1786.0x	752.8x
jsoncpp	450	1.6s	5.9s	3.7x	478.4s	299.0x	81.1x	1395.4s	872.1x	236.5x
openthread	268	0.9s	3.8s	4.2x	18.2s	20.2x	4.8x	204.2s	226.9x	53.7x
openssl	1577	11.3s	88.4s	7.8x	43255.3s	3827.9x	489.3x	215200.1s	19044.3x	2434.4x
mbedtls	491	1.6s	18.1s	11.3x	4146.9s	2591.8x	229.1x	9532.2s	5957.6x	526.6x
sqlite3	5253	19.7s	257.7s	13.1x	N/A	N/A	N/A	46465.7s	2358.7x	180.3x
curl	1343	7.1s	35.0s	4.9x	398.8s	56.2x	11.4x	4494.8s	633.1x	128.4x
Geomean				9.2x		589.2x	**62.0x**		3407x	**371.1x**

Table 6: Statistics of the solved constraints

	SymSan	SymCC
Number of processed constraints	15,860,404	7,102,939
Number of satisfied nested constraints	5,022,332 (31.67%)	2,294,057 (32.30%)
Timeout nested constraints	12013 (0.076%)	1140 (0.016%)

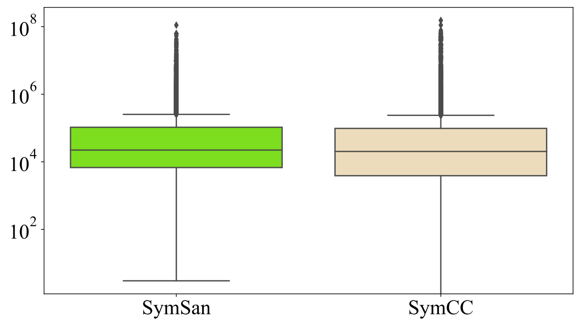

Figure 11: Maximum number of AST nodes tracked by SymSan and SymCC

Table 7: Mean bug survival times—both **R**eached and **T**riggered—over a 24-hour period, in **s**econds, **m**inutes, and **h**ours. Bugs are sorted by "difficulty" (mean times).

	symcc		symsan		symsan_sec			symcc		symsan		symsan_sec	
Bug ID	R	T	R	T	R	T	Bug ID	R	T	R	T	R	T
AAH037	10.0s	25.00s	10.00s	25.00s	10.00s	25.00s	AAH041	15.00s	25.50s	15.00s	27.0s	15.00s	26.50s
AAH003	10.00s	1.08m	10.00s	15.00s	10.00s	15.00s	JCH207	10.00s	1.67m	10.00s	2.65m	10.00s	2.58m
AAH056	15.00s	8.00m	15.00s	7.68m	15.00s	7.63m	AAH015	16.67m	41.42m	17.12m	56.25m	17.21m	58.20m
AAH055	15.00s	15.68m	20.00s	2.53m	20.00s	2.51m	AAH020	5.00s	11.63h	10.00s	2.40h	10.00s	2.89h
MAE016	24.00h	24.00h	24.00h	24.00h	24.00h	24.00h	AAH052	15.00s	6.36h	15.00s	7.33m	15.00s	7.26m
AAH032	15.00s	23.72h	15.00s	11.11h	15.00s	8.00h	MAE008	24.00h	24.00h	24.00h	24.00h	24.00h	24.00h
AAH022	15.07m	16.86h	17.12m	20.49h	17.21m	17.07h	MAE014	24.00h	24.00h	24.00h	24.00h	24.00h	24.00h
JCH215	1.85h	22.09h	3.33h	14.51h	5.10h	13.65h	AAH017	11.05h	11.18h	7.63h	7.69h	5.84h	5.84h
JCH232	24.00h	24.00h	7.25h	10.71h	9.41h	15.88h	AAH014	12.87h	12.87h	20.56h	20.38h	23.19h	23.19h
JCH201	15.00s	16.08h	15.00s	1.07h	15.00s	1.06h	AAH007	15.00s	15.52m	15.00s	10.15m	15.00s	12.18m
AAH008	15.00s	24.00h	15.00s	22.07h	15.00s	22.88h	AAH045	20.00s	24.00h	20.00s	24.00h	20.00s	24.00h
AAH013	24.00h	24.00h	23.31h	24.00h	24.00h	24.00h	AAH024	15.00s	24.00h	15.00s	24.00h	15.00s	24.00h
JCH209	24.00h	24.00h	23.47h	23.47h	21.67h	21.67h	MAE115	15.00s	13.95h	15.00s	16.91h	15.00s	15.37h
AAH026	15.00s	24.00h	15.00s	24.00h	15.00s	24.00h	AAH001	15.00s	57.04m	15.00s	8.12m	15.00s	29.00s
MAE104	24.00h	22.48h	15.00s	17.70h	15.00s	17.37h	AAH010	1.76h	24.00h	7.89h	24.00h	23.42h	23.19h
AAH016	24.00h	24.00h	24.00h	24.00h	24.00h	24.00h	JCH226	24.00h	24.00h	16.07h	24.00h	22.53h	24.00h
JCH228	24.00h	24.00h	11.75h	24.00h	13.02h	24.00h	AAH035	15.00s	24.00h	15.00s	24.00h	15.00s	24.00h
JCH212	15.00s	24.00h	15.00s	24.00h	15.00s	24.00h	AAH025	24.00h	24.00h	22.64h	24.00h	24.00h	24.00h
AAH053	24.00h	24.00h	24.00h	24.00h	24.00h	24.00h	AAH042	45.00s	24.00h	45.00s	24.00h	45.00s	24.00h
AAH048	20.00s	24.00h	20.00s	24.00h	20.00s	24.00h	AAH049	15.00s	24.00h	15.00s	24.00h	15.00s	24.00h
AAH043	20.00s	24.00h	21.87h	24.00h	21.73h	24.00h	JCH210	35.00s	24.00h	60.00s	24.00h	55.00s	24.00h
AAH050	30.00s	24.00h	30.00s	24.00h	30.00s	24.00h	AAH054	10.00s	24.00h	10.00s	24.00h	10.00s	24.00h
MAE105	15.00s	24.00h	15.00s	24.00h	15.00s	24.00h	AAH011	10.00s	24.00h	10.00s	24.00h	10.00s	24.00h
AAH005	15.00s	24.00h	15.00s	24.00h	15.00s	24.00h	JCH202	15.00s	24.00h	15.00s	24.00h	15.00s	24.00h
MAE114	15.00s	24.00h	15.00s	24.00h	15.00s	24.00h	AAH029	15.00s	24.00h	15.00s	24.00h	15.00s	24.00h
AAH034	15.00s	24.00h	15.00s	24.00h	15.00s	24.00h	AAH004	15.00s	24.00h	15.00s	24.00h	15.00s	24.00h
MAE111	15.00s	24.00h	15.00s	24.00h	15.00s	24.00h	AAH059	15.00s	24.00h	15.00s	24.00h	15.00s	24.00h
JCH204	20.00 s	24.00h	30.00s	24.00h	30.00s	24.00h	AAH031	25.00s	24.00h	55.00s	24.00h	1.02m	24.00h
AAH051	20.00s	24.00h	25.00s	24.00h	25.00s	24.00h	MAE103	20.00s	24.00h	20.00s	24.00h	20.00s	24.00h
JCH214	35.50s	24.00h	35.00s	24.00h	35.00s	24.00h	JCH220	6.01h	24.00h	4.80h	24.00h	5.19h	24.00h
JCH229	2.32h	24.00h	5.39h	23.61h	5.29h	24.00h	AAH018	1.24h	24.00h	7.20h	24.00h	6.60h	24.00h
JCH230	6.05h	24.00h	8.07h	24.00h	8.59h	24.00h	AAH047	25.00s	24.00h	25.00s	24.00h	25.50s	24.00h
JCH233	7.10h	24.00h	7.94h	24.00h	10.81h	24.00h	JCH223	12.82h	24.00h	8.04h	24.00h	8.64h	24.00h
JCH231	14.64h	24.00h	8.11h	24.00h	8.70h	24.00h	MAE006	24.00h	24.00h	24.00h	24.00h	24.00h	24.00h
MAE004	24.00h	24.00h	24.00h	24.00h	24.00h	24.00h	JCH222	18.08h	24.00h	11.59h	24.00h	16.26h	24.00h
AAH009	24.00h	24.00h	23.46h	24.00h	24.00h	24.00h	JCH227	24.00h	24.00h	23.17h	24.00h	20.46h	24.00h
JCH219	24.00h	24.00h	24.00h	24.00h	22.83h	24.00h	JCH216	24.00h	24.00h	24.00h	24.00h	24.00h	24.00h