UNIVERSITY OF CALIFORNIA
RIVERSIDE

Building an Efficient Concolic Executor

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

by

Ju Chen

June 2022

Dissertation Committee:

Professor Chengyu Song, Co-Chairperson
Professor Heng Yin, Co-Chairperson
Professor Rajiv Gupta
Professor Zhiyun Qian

The Dissertation of Ju Chen is approved:

Committee Co-Chairperson

Committee Co-Chairperson

University of California, Riverside

Acknowledgments

I am grateful to my advisor, Professor Chengyu Song, and Professor Heng Yin. Without their help, I would not have been here. Their knowledge, enthusiasm, and continuous support are the key to my Ph.D. study. Besides, my sincere gratitude goes to the rest of my Ph.D. dissertation committee: Professor Zhiyun Qian and Professor Rajiv Gupta for their insightful questions and comments. I would like to thank our collaborator, Professor Byoungyoung Lee, for his insightful suggestions. I also want to thank my fellow Jinghan, Jie, Mingjun, Haochen, Sheng, Zhenxiao, Zixiang, Wei, Lian, and Yu, for their help during my hard times. This dissertation includes previously published materials entitled "SymSan: Time and Space Efficient Concolic Execution via Dynamic Data-flow Analysi" published in the USENIX Security Symposium, 2022, and "JIGSAW: Efficient and Scalable Path Constraints Fuzzin" published in the IEEE Symposium on Security and Privacy, 2022.

To my wife for all the support.

ABSTRACT OF THE DISSERTATION

Building an Efficient Concolic Executor

by

Ju Chen

Doctor of Philosophy, Graduate Program in Computer Science
University of California, Riverside, June 2022
Professor Chengyu Song, Co-Chairperson
Professor Heng Yin, Co-Chairperson

Concolic execution is a powerful program analysis technique for systematically exploring execution paths. It is especially good at exploring paths that are guarded by complex and tight branch predicates. However, concolic execution faces scalability issues that prevent concolic execution from being adopted widely in practice. This thesis tries to reduce the overhead from two major sources in the concolic execution.

The first source of overhead is constraints collecting. This thesis presents a novel design that models concolic execution as a special form of dynamic data-flow analysis. Therefore, the concolic executor can be implemented by leveraging the existing highly-optimized data-flow analysis framework. The evaluation results show that the new design can significantly improve the efficiency of the concolic execution and reduce its memory consumption.

The second source of overhead is path constraints solving. This thesis presents a novel way to flip branches efficiently. The insight is that constraints collected by concolic executors are pure and straight-line functions that make pure functions an ideal target for

evaluating newly generated test inputs. Concretely, by converting constraints to native programs (using JIT compilation), more than 1 million inputs can be tested per second. The evaluation results show that this new design can significantly outperform existing SMT solvers.

In summary, this thesis shows that with the improvements in constraints collecting and constraints solving, a concolic executor can be built more efficiently than the existing tools and is helpful in coverage-guided software testing.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

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.

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 [13]. 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).

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, the concolic executor checks the feasibility of the branch target opposite to the concrete direction; if feasible, it generates a corresponding input.

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 two performance bottlenecks: constraints collecting and constraints solving.

For constraint collecting, Yun *et al.* [77] 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 instruction-level [52], Poeplau and Francillon proposed using IR-level instrumentation to (i) avoid symbolic emulation of instructions and (ii) retain the simplicity of symbolic constraints [53]. As a result, their tool

SymCC performs significantly faster than both IR-less QSYM [77] and IR-based KLEE [11]. Despite recent improvements on symbolic execution, the state-of-the-art symbolic executors still impose a significant performance and memory overhead compared to native execution.

For constraint solving, Poeplau and Francillon [53] reported that constraint solving contributes up to 93.3% of the concolic execution's performance overhead. We also have a similar observation when testing programs such as `objdump` `libjpeg`, `vorbis` and `woff2`. As a concrete example, it takes about 24 seconds to symbolically execute `objdump` on 560 inputs without solving, while it takes 12 hours to finish executing when solving is turned back on.

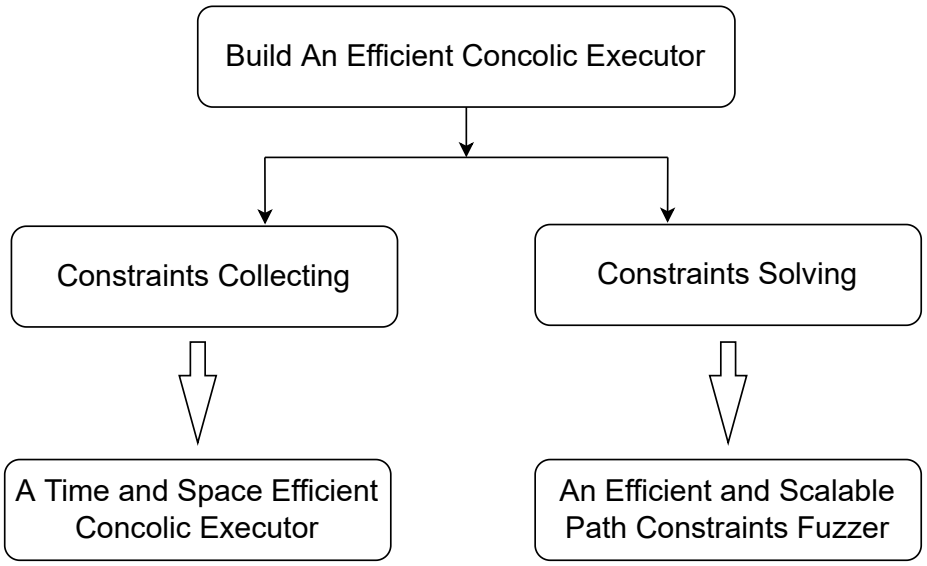

Figure 1.1: Thesis Overview

1.1 Thesis Statement

My thesis aims to make concolic execution more efficient in open-sourced programs. To this end, as illustrated in Figure 1.1, this project aims to tackle two interconnected tasks by proposing two novel techniques.

- **An efficient concolic executor.** This task aims to improve the efficiency of concolic execution. To this end, we propose SymSan, a time and space efficient concolic executor built based on the high-optimized data-flow analysis framework.

- **An efficient path constraints solver.** This task aims to address the bottleneck of constraints solving. To this end, we propose JIGSAW, an efficient and scalable path constraints solver. JIGSAW finds feasible inputs for a path constraint by (just-in-time) compiling the constraint to a pure native function(s) and performing gradient descent search on the JIT'ed function(s).

Chapter 2

Background

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.

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 [13].

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.*, $\epsilon =$ `true` for the `if then` branch or $\epsilon =$ `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 $PC \wedge \epsilon$ and $PC \wedge \neg\epsilon$ are satisfiable). For each feasible direction, the symbolic executor updates its path constraint PC by adding the constraint ($\epsilon = \texttt{true}$ or $\epsilon = \texttt{false}$) 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).

2.1.1 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.

2.1.2 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.* [77] 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 instruction-level [52], Poeplau and Francillon proposed using IR-level instrumentation to (i) avoid symbolic emulation of instructions and (ii) retain the simplicity of symbolic constraints [53]. As a result, their tool SymCC performs significantly faster than both IR-less QSYM [77] and IR-based KLEE [11].

2.2 Automated Test Generation

Testing is an important and effective way to detect software bugs. However, manually generated test cases are usually biased towards normal or expected inputs so they do not provide enough coverage, especially for corner cases. As a result, simply testing software with random inputs is enough to generate many crashes [44], most of which are exploitable. Automated test generation aims to generate test cases to cover as much code as possible. Fuzzing and symbolic execution are the two most popular automated test generation techniques.

Fuzzers create inputs in a generative manner or mutational manner. Generative fuzzers can be grammar guided [28, 23, 1, 58, 47] or learning based [71, 56, 31]. Mutational fuzzers generally adopt two genetic operations: random mutation and crossover [44, 78, 33, 62]. The first generation of fuzzers was blackbox fuzzers [44, 23, 1], which just create random test inputs. While they have successfully found many bugs, most of those bugs are shallow; once fixed, these fuzzers cannot go deeper and cover more code/states. The reason is that blackbox fuzzers are aimless so they can easily generate lots of redundant test cases. To solve this problem, greybox (*a.k.a.*feedback-guided) fuzzers were invented [78, 33, 62, 38, 51, 5, 69, 6, 2, 75, 20, 25, 16, 17, 4, 42, 76]. By using lightweight instrumentation to collect limited runtime information (*e.g.*, branch coverage), greybox fuzzers can *measure* the progress they have made and steadily progress towards their goals [48].

White-box fuzzers and symbolic/concolic executors [30, 19, 12, 77, 61, 13, 10, 29, 11, 14, 53, 54, 9] generate new input test cases more systematically. They treat the test input as a sequence of symbolic bytes. When executing the target program, a symbolic execution engine maintains (1) a symbolic state σ, that maps program variables to symbolic expressions and (2) a set of quantifier-free first-order formulas over symbolic expressions that are imposed by conditional branches (*a.k.a.*path constraints) [13]. Whenever the execution engine encounters an uncovered branch, it will query an SMT solver for the satisfiability of that branch's predicate under current path constraints. If the branch predicate is satisfiable, it asks the SMT solver to return a model for the relevant inputs bytes and generates a new test input that should be able to cover that branch.

2.2.1 Efficiency of Test Generation

Since we only have limited resources (CPU, memory, and time), the most important metric for measuring an automated test generation technique is its efficiency, i.e., how much coverage can it achieve with the limited resources. The first component that has a huge impact on efficiency is state/path scheduling. For fuzzers, since each testcase represents an execution path, testcase scheduling is the same as path scheduling. A basic observation is that if opposite branches along a path have already been covered or are hard/infeasible to flip, then spending more time on this path will not give any reward (new coverage). This scheduling problem can be generally modeled as a multi-armed bandit (MAB) problem [15, 76] and numerous scheduling algorithms have been proposed to improve the efficiency of fuzzers [5, 6, 39, 26, 57, 4, 76, 72, 70].

Once a path is scheduled, the next important factor that affects the efficiency is the speed to flip an uncovered branch. The branch flipping problem is a typical search problem: how to find an input that can satisfy the branch predicate and additional path constraints that must be satisfied to reach this branch [17]. The efficiency of this step depends on two factors. The first factor is the search algorithm. Off-the-shelf fuzzers [78, 33, 62] do not pay much attention to this problem and rely on a random search. As a result, their search is aimless and usually faces difficulties when trying to flip branches with tight constraints (*e.g.*, magic number check). To overcome this limitation, researchers have proposed numerous heuristics [57, 2, 50, 73]. A more general solution is to measure progress and perform a directed search, such as splitting branches [37], using gradient-guided search [16, 17, 64, 65], binary search [20], genetic algorithms [25], or simulated annealing [67]. For complex path

constraints, the most efficient way so far is to use an SMT solver, which applies a large set of sophisticated heuristics to transform/rewrite the constraints into simpler ones, then searches for a satisfying solution. Modern SMT solvers usually leverage two main solving strategies for path constraints that are in the quantifier-free theories of bit-vectors and arrays: (1) bit-blasting, which reduces the constraints into a corresponding SAT (boolean satisfiability) problem, then queries an efficient SAT solver to find a solution; (2) local search, which transforms the constraints into an objective function and applies optimization techniques to find a solution. Recent research also shows that employing the aforementioned fuzzing heuristics can be beneficial [40, 49, 8]. Note that the focus of this work is *not* on improving the search heuristics, but on improving the throughput; and our approach can be combined with any fuzzing- or local-search-based heuristics.

The second factor that affects the efficiency of branch flipping is the number of new inputs that can be tried in a unit of time. The more inputs a fuzzer can try, the faster it can find a satisfying input. For this reason, efforts have also been made to improve the throughput of fuzzers. For example, AFL [78] uses `fork_server` to avoid initialization overhead. kAFL [60] avoids instrumentation by using a hardware trace collector. Firm-AFL [79] avoids expensive whole-system emulation through augmented user-mode emulation. Xu *et al.* [74] designed three new operating system (OS) primitives to improve the scalability of parallel fuzzing on multi-core machines. Nyx [59] employs a fast virtual machine reset technique. By evaluating with JIT'ed path constraints, our approach can significantly improve the search throughput.

Previously, the major drawback of symbolic execution has been that collecting symbolic constraints is very slow [77], so the overall branch flipping efficiency is not as good as greybox fuzzers. However, recent advances in constraints collection have largely reduced this overhead [77, 53, 54, 9].

Chapter 3

SymSan: Reducing The Overhead of Managing Symbolic Expressions

3.1 Motivation

Despite recent improvements on 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 [77])

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.

3.2 Design

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

Overview. Similar to existing instrumentation-based concolic executors like SymCC [53], SymSan performs compile-time instrumentation to insert the logic for introducing, propa-

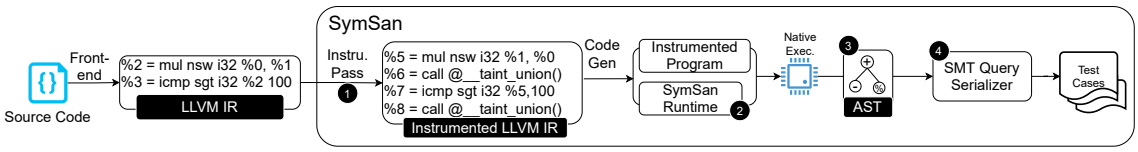

Figure 3.1: The overall design of SymSan

gating, and checking symbolic expressions. The overall architecture of SymSan is shown in Figure 3.1. ❶ SymSan takes a compiled LLVM IR as input and instruments the code via a compiler pass. SymSan's run-time ❷ 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) ❸ of all program variables is then populated according to the symbolic execution policy. At points of interest (*e.g.*, conditional branches), ❹ 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 3.2 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 3.2 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:

```
; bool example(int *a, int b) {
;   return (*a) * b > 100;
; }
define i1 @example(i32* %0, i32 %1) {
  %3 = load i32, i32* %0
  %4 = mul nsw i32 %3, %1
  %5 = icmp sgt i32 %4, 100
  ret i1 %5

define i1 @"dfs$example"(i32* %0, i32 %1) {
  ; load taint labels for the arguments
  %3 =load i32,getelementptr(@__dfsan_arg_tls, 0) ; arg0
  %4 =load i32,getelementptr(@__dfsan_arg_tls, 1) ; arg1
  ; load taint label from shadow memory
  %5 = ptrtoint i32* %0 to i64  ; get shadow_addr(%0)
  %6 = and i64 %5, -123145302310913
  %7 = mul i64 %6, 4
  %8 = inttoptr i64 %7 to i32*
  %9 = call i32 @__taint_union_load(i32* %8, i64 4)
  ; load concrete value
  %10 = load i32, i32* %0
  ; concrete execution
  %11 = mul nsw i32 %10, %1
  ; create a new label to represent (*a) * b
  %12 = zext i32 %11 to i64  ; extend
  %13 = zext i32 %1 to i64
  %14 = call i32 @__taint_union(
        i32 %9, i32 %4,  ; symbolic operands
        i16 MUL,  ; operator
        i8 32,    ; operand size in bits
        i64 %12, i64 %13 ; concrete operands
        )
  ; concrete execution
  %15 = icmp sgt i32 %11, 100
  ; create a new label to represent (*a) * b > 100
  %16 = zext i32 %15 to i64
  %17 = call i32 @__taint_union(
        i32 %14, ; symbolic left operand
        i32 0,   ; zero label for concrete right operand
        i16 ICMP_LARGER_THAN,  ; operator
        i8 32,  ; operand size
        i64 %15, i64 100  ; concrete operands
        )
  ; store the label of the return value
  store i32 %17, @__dfsan_retval_tls
  ret i1 %15
}
```

Figure 3.2: A running example illustrating how SymSan instrument the target program.

- **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 subsection 3.2.2), 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 [63]), 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).

3.2.1 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 [53].

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 (subsection 3.2.2).

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 3.2, the first way is through the per-thread thread local storage (TLS). Accessing TLS is very fast, which 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

uses 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 [63]. 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 3.2, 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 provides optimal accesses 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 for storing and retrieving symbolic expressions. Moreover, SymSan uses a more concise representation for symbolic expressions.

3.2.2 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 [53] 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 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 node) 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 [6] 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 3.5, 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 3.3: AST node of SymSan.

AST Nodes. Figure 3.3 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 into 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 (subsection 3.2.3). 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 3.1) 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.

3.2.3 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 a 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 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 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.

3.2.4 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 section 3.4 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 key words, 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 to the bytes in the second buffer).

3.3 Implementation

In this section, we reveal some implementation details of our SymSan. SymSan is implemented based on the data-flow sanitizer (DFSAN) [68], which is part of the LLVM compiler toolchain.

Table 3.1: 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 special memory layout. Table 3.1 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 fair comparison with SymCC [53] and SymQEMU [54], which also do not support non-integer operations. We also do not support 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 a 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 [77] and SymCC [53] 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 [77] 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 standard C library, we implemented custom wrapper functions to propagate labels. For standard C++ library, we instrumented `libc++` from LLVM.

3.4 Evaluation

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

- **RQ1: Time efficiency.** Does SymSan impose less runtime 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 testcases 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 [53] and SymQEMU [54]. We believe the comparison with SymCC is especially meaningful as both CEs perform compile-time instrumentation at the LLVM IR level, as use Z3 as the constraint solver. For hybrid fuzzing, we include the state-of-the-art fuzzer AFL++ [24] for comparison.

3.4.1 Dataset

Standard Benchmark. We choose nbench [43] to evaluate the instrumentation overhead of SymSan and baseline CEs. 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 evaluation of state-of-the-art CEs [77, 53, 54]. 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 20 real-world programs shown in Table 3.3. 16 programs are from Google's Fuzzbench [34], 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 [64] for better reproducibility.

3.4.2 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 of symbolic expressions.
- **Constraint solving.** The overhead from consulting an SMT solver.

Pure Concrete Execution

We ran programs in nbench [43] 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

Table 3.2: Performance results for pure concrete execution on NBENCH

Tests (Iterations/s)	Native	SymSan	SymCC	SymQEMU
NUMERIC SORT	1785.3	169.34	29.025	15.153
STRING SORT	2154.1	703.02	1.1445	1.2757
BITFIELD	8.68e+08	3.19e+07	9.79e+06	5.34e+06
FP EMULATION	828.42	17.114	16.382	4.9317
FOURIER	71570	2811.6	8901.6	261.54
ASSIGNMENT	66.037	3.7029	0.8271	0.28006
IDEA	14819	529.04	241.16	85.503
HUFFMAN	6112.6	279.63	75.263	32.832
NEURAL NET	129.7	6.1487	0.61978	0.17407
LU DECOMPOSITION	3323.3	145.08	24.214	8.4286
Score Index				
Memory Index	67.104	5.881	0.283	0.167
Integer Index	47.344	2.002	0.754	0.298
Floating-point Index	79.270	3.434	1.292	0.184

backend. In this way, we can measure the instrumentation overhead of each concolic executor. Table 3.2 reports the results. Compared to native execution, SymSan is 7.7× slower on memory index, 17.1× slower on integer index, and 5.45× slower on floating-point index. SymCC and SymQEMU are much slower than SymSan. SymCC is 215.4× slower on memory index than native execution, 57.1× slower on integer index, and 64.5× slower on floating-point index. SymQEMU is 332.7×, 128.7×, and 356.9× slower than native execution, respectively. We also noticed that SymSan performed much better than SymCC on memory index. We believe this is due to the direct-mapping-based shadow memory scheme used by SymSan.

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

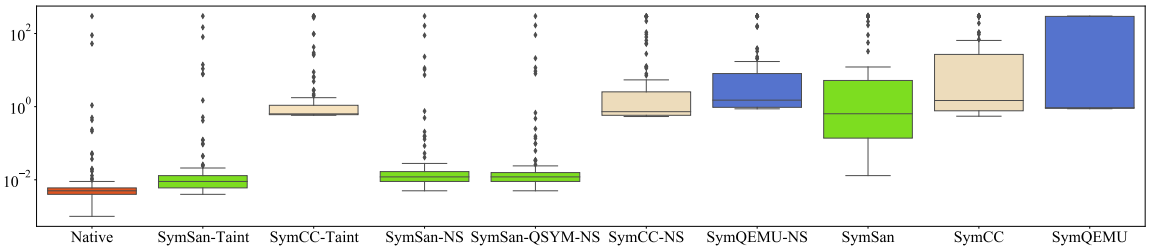

Figure 3.4: Execution time of 102 CGC challenge binaries

SymSan, and a new expression pointer (casted from an increasing integer) for SymCC. This comparison shows the benefit of SymSan's shadow memory implementation.

CGC. Following the same procedure as previous papers [77, 53, 54], we used the first PoV input to test each CGC challenge. We enforced the same 5-minute timeout for each execution as [77] for easier comparison with number reported in previous papers. The results are shown in Figure 3.4. Compare to native execution, SymSan (`SymSan-Taint`) has 1.3 times slowdown, and SymCC (`SymCC-Taint`) has 4.9 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 3.5. Overall, SymSan (`SymSan-Taint`) introduces 6.0 times overhead comparing to the native execution, while SymCC (`SymCC-Taint`) introduces 67.2 times overhead.

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

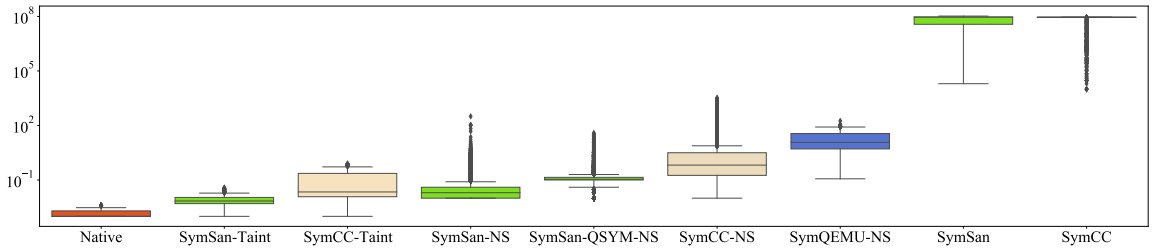

Figure 3.5: Execution time for the real-world programs.

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 subsection 3.2.3 is in Appendix. Overall, constraint deduplication improved the performance by 20% and load/store simplification added another 120% 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 3.4. SymSan (`SymSan-NS`) is 1.3 times slower than the native execution, and SymSan with QSYM backend (`SymSan-QSYM-NS`) is 1.4 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 7.2 and 9.9 times slower than SymSan respectively.

Real-world applications. The distribution for inputs that did not timeout is shown in Figure 3.5. 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, SymSan (`SymSan-NS`) introduces 9.2× overhead comparing to the native execution, while SymCC (`SymCC-NS`) and SymQEMU (`SymQEMU-NS`) introduce 589.2× to 3407× overhead respectively. As a result, SymSan achieves 62.0× performance speedup over SymCC and 371.1× 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 (15.4× speedup), and only imposes a small overhead over `SymSan-Taint`.

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 3.4.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 time, which was also reported in previous work [53].

Real-world applications. For real-world application, we placed a 90-second timeout for each execution. Otherwise, the experiments cannot be completed in a reasonable time. Note

Table 3.3: Execution time of concolic execution engines with solving (in seconds).

Program	#seeds	Total Execution Time (sec)			Basic Block Coverage		
		SymSan	SymCC	SymQEMU	SymSan	SymCC	SymQEMU
readelf	604	27,712	41,695	49,947	7,938	6,067	3,807
objdump	560	43,612	44,052	47,627	4,853	4,668	4,528
nm	249	6,106	14,127	18,628	3,492	2,746	2,755
size	207	3,517	8,920	15,976	2,647	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	4,452	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	1,251	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	4,177	4,145	4,140
sqlite3	5253	111,029	N/A	421,947s	32,765	N/A	36,124
curl	1343	501	1,312	102,180s	13,171	13,122	13,140

that placing a timeout for each concolic execution is a common practice adopted by both SymCC and SymQEMU. The results are shown in Table 3.3. The execution time distribution for inputs that did not timeout is shown in Figure 3.5. 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.

3.4.3 Memory Consumption

In this section, we evaluated the memory usage by SymSan and compared 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 3.6. As we can see, SymSan introduces much smaller memory overhead than SymCC (3.5× vs. 98.2×). The result also shows that our AST table is more memory efficient than the QSYM backend.

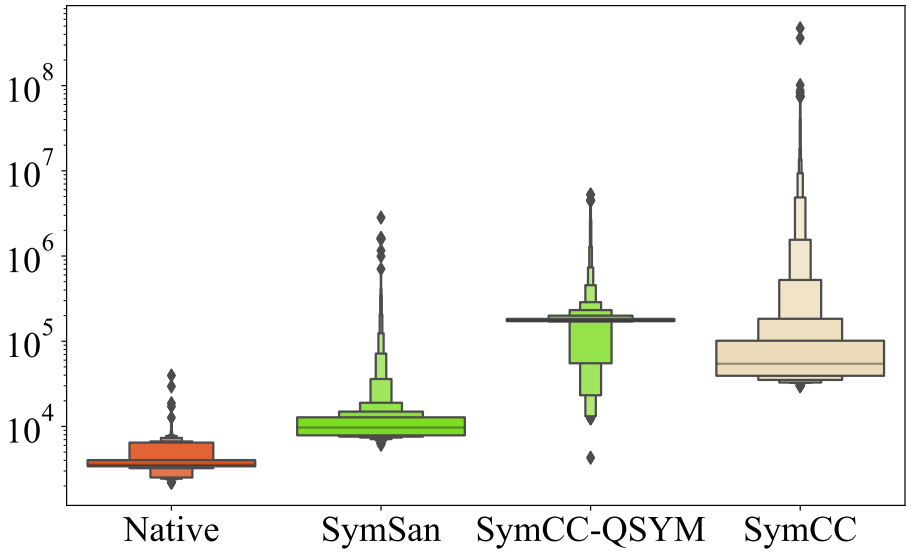

Figure 3.6: The peak resident size for each concolic execution without solving.

3.4.4 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 the method introduced by Yun *et al.* [77]. 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 of A and B is then calculated as below as per [77] (when $A! = B$):

$$d(A,B) = \frac{|A-B|-|B-A|}{|(A \cup B)-(A \cap B)|}$$

The score will be in range of $[-1.0, 1.0]$, where 1.0 means SymSan not only covers all paths that are covered by other concolic executor but also covers some unique paths. Our results is visualized in Figure 3.7. As we can see, SymSan has the similar code coverage as SymCC, it covers slightly more than SymCC in 83 programs while covers less in 19 programs.

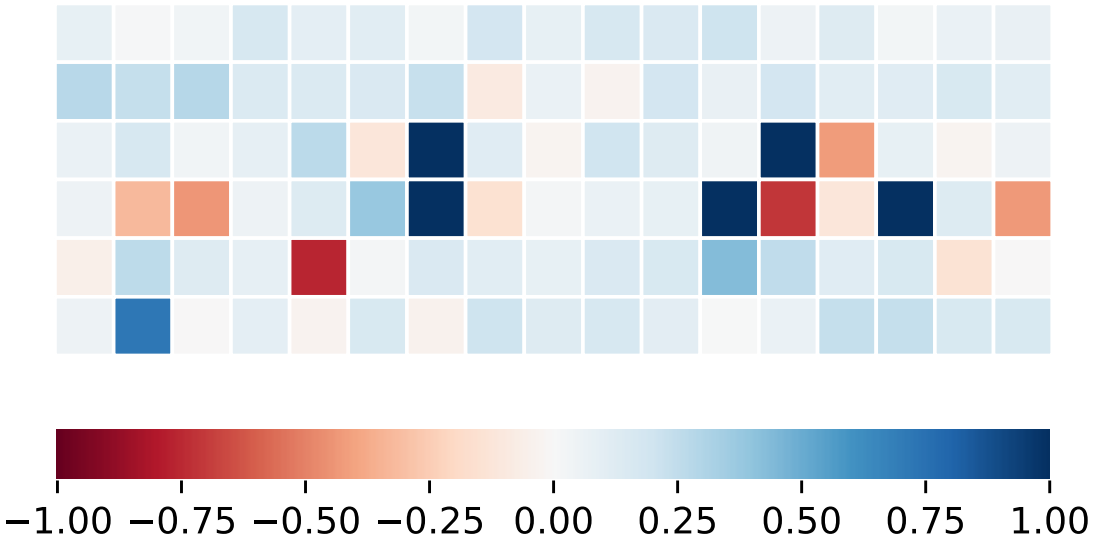

Figure 3.7: Coverage score comparing SymSan and SymCC per tested program (102 CGC challenge binaries in total).

Real-world Applications. For each real-world application, We measured the basic-block coverage for all generated inputs using SanitizerCoverage [41]. The results is in Table 3.3. 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 binary-based concolic executor and can handle external libraries better than SymSan. In 11 out of 20 programs tested, SymSan covers more than SymCC and SymQEMU. In the rest of 11 programs, SymSan covers slightly less than SymCC and SymQEMU.

3.4.5 Hybrid Fuzzing

In this evaluation, we plugged SymSan in 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 [34]. We use AFL++ (commit `70bf4b4` with the default build and fuzz options[1]) 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 [53], SymQEMU [54], and Fuzzolic [9] based on their publicly available experiment report[2]. 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 3.4. 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 [53], and QSYM [77] used the same hybrid fuzzing configuration as described in the QSYM's tutorial: a concolic executor paired with

[1] `https://github.com/google/fuzzbench/blob/master/fuzzers/aflplusplus/fuzzer.py`
[2] `https://www.fuzzbench.com/reports/experimental/2021-07-03-symbolic/index.html`

two AFL (version 2.56b) instances, one master and one slave. In addition, the concolic executor has a 90 seconds timeout for executing each seed. For each concolic executor/fuzzer, we executed 10 fuzzing trials, each for 24 hours. To ensure fair comparison, we uses the Fuzzbench's configuration to run each fuzzer/concolic executor in a docker container with 1 physical CPU-core assigned.

The result is shown in Figure 3.8. SymSan can achieve higher final coverage than other tools on the four programs from binutils. For rest three programs, SymSan performs similarly to another 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.

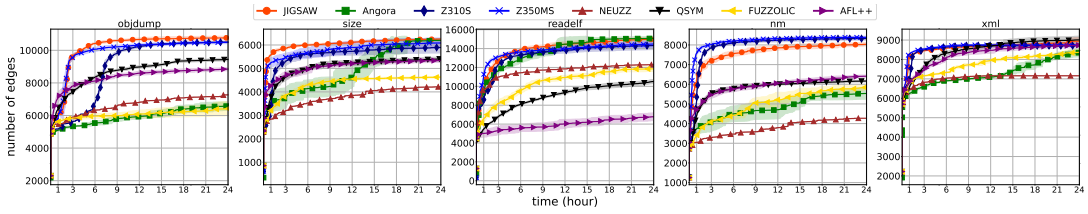

Figure 3.8: Edge coverage growth over time for local fuzzing.

3.4.6 Security Implications

Recent research has show a strong correlation between a testing tool's ability to achieve code coverage and its ability to find bugs [7]. Similarly, recent research also showed

Table 3.4: Comparing SymSan with other state-of-the-art symbolic executors based on their publicly available Fuzzbench results.

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**

that leveraging symbolic execution to solve bug triggering constraints can also improve a hybrid fuzzer's ability to find bugs [18]. Based on these observations, we expect that SymSan can also help on 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 [35], 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 [18]), and (3) `symccafl` (SymCC with AFL). The full results are shown in Table 3.5. The present numbers are average over 10 trials. Following are a few highlights.

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

Table 3.5: 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	25.50s	10.00s	25.00s	10.00s	25.00s	AAH041	15.00s	30.00s	15.00s	26.50s	15.00s	26.50s
AAH003	10.00s	1.58m	10.00s	15.00s	10.00s	15.00s	JCH207	10.00s	1.62m	10.00s	2.34m	10.00s	1.37m
AAH056	15.00s	17.80m	15.00s	9.07m	15.00s	7.81m	AAH015	15.07m	1.02h	17.12m	59.22m	17.21m	58.20m
AAH055	15.00s	2.44h	20.00s	13.33m	20.00s	15.72m	AAH020	5.00s	11.22h	10.00s	2.38h	10.00s	2.89h
MAE016	24.00h	24.00h	24.00h	24.00h	24.00h	24.00h	AAH052	15.00s	5.28m	15.00s	4.78h	15.00s	4.61h
AAH032	15.00s	12.95h	15.00s	7.91h	15.00s	8.00h	MAE008	24.00h	24.00h	24.00h	24.00h	24.00h	24.00h
AAH022	15.07m	15.25h	17.12m	22.22h	17.21m	17.07h	MAE014	24.00h	24.00h	24.00h	24.00h	24.00h	24.00h
JCH215	1.85h	18.08h	3.33h	14.51h	5.10h	13.65h	AAH017	9.92h	9.92h	7.63h	7.63h	5.84h	5.84h
JCH232	21.81h	21.81h	7.25h	10.71h	9.41h	15.88h	AAH014	10.68h	10.68h	20.56h	20.56h	23.19h	23.19h
JCH201	15.00s	14.27h	15.00s	24.00h	15.00s	24.00h	AAH007	15.00s	23.12m	15.00s	10.15m	15.00s	12.18m
AAH008	15.00s	23.43h	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	23.31h	24.00h	24.00h	AAH024	15.00s	24.00h	15.00s	24.00h	15.00s	24.00h
JCH209	24.00h	24.00h	24.00h	24.00h	21.67h	21.67h	MAE115	15.00s	10.13h	15.00s	18.76h	15.00s	21.58h
AAH026	15.00s	24.00h	15.00s	24.00h	15.00s	24.00h	AAH001	15.00s	14.58h	15.00s	8.12m	15.00s	29.00s
MAE104	24.00h	24.00h	15.00s	16.46h	15.00s	18.16h	AAH010	4.76h	24.00h	7.89h	22.09h	23.42h	24.00h
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	22.66h	23.80h	11.75h	24.00h	13.02h	24.00h	AAH035	19.00s	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	22.64h	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	25.00s	24.00h	21.87h	24.00h	21.73h	24.00h	JCH210	32.50s	24.00h	60.00s	24.00h	55.00s	24.00h
AAH050	29.00s	24.00h	30.00s	24.00h	30.00s	24.00h	AAH054	10.00s	24.00h	10.00s	24.00h	10.00s	24.00h
MAE105	10.00s	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	20.00s	24.00h	15.00s	24.00h	15.00s	24.00h	AAH059	20.00s	24.00h	15.00s	24.00h	15.00s	24.00h
JCH204	20.00s	24.00h	30.00s	24.00h	30.00s	24.00h	AAH031	25.00s	24.00h	55.00s	24.00h	1.02m	24.00h
AAH051	30.00s	24.00h	25.00s	24.00h	25.00s	24.00h	MAE103	31.00s	24.00h	20.00s	24.00h	20.00s	24.00h
JCH214	35.00s	24.00h	35.00s	24.00h	35.00s	24.00h	JCH220	2.28h	24.00h	4.80h	24.00h	5.19h	24.00h
JCH229	2.32h	24.00h	5.39h	23.61h	5.29h	24.00h	AAH018	1.85h	24.00h	7.20h	24.00h	6.60h	24.00h
JCH230	5.57h	24.00h	8.07h	24.00h	8.59h	24.00h	AAH047	25.00s	24.00h	25.00s	24.00h	25.50s	24.00h
JCH233	5.17h	24.00h	7.94h	24.00h	10.81h	24.00h	JCH223	10.60h	24.00h	8.04h	24.00h	8.64h	24.00h
JCH231	10.62h	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	20.82h	24.00h	11.59h	24.00h	16.26h	24.00h
AAH009	23.42h	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

takes 14.58 hours to trigger the bug, and the fastest mutational fuzzer Honggfuzz uses 17.7 hours [35].

- **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 hours. `symsan` uses 7 hours to trigger this bug. In comparison, `symccafl` takes 9.9 hours to trigger the bug, and the fastest mutational fuzzer `moptafl` takes 5.2 hours [35].
- **AAH055 (CVE-2016-2108)** is a out-of-bound read issue in `openssl`. Both `symsan` and `symsan_sec` can trigger this bug, using 13 minutes and 15 minutes, respectively. However, it takes 2.44 hours for `symccafl` to trigger the bug.

Chapter 4

JIGSAW: Speeding Up Branch Flipping

4.1 Overview

4.1.1 Insight

Our design goal is to push the search throughput (*i.e.*, the number of test inputs got evaluated per unit time) to the next level. To achieve this goal, we leverage an important insight: *path constraints collected by symbolic executors are pure and straight-line functions*. Similar to a mathematical function, a pure function always returns the same value on the same inputs (*i.e.*, there is no hidden dependencies over global states) and has no side-effect (*i.e.*, will not affect global states). This makes pure functions an ideal target for evaluating newly generated test inputs, because **P1**. no side-effect means no need to perform expensive state reset (*e.g.*, invoking `fork()`). **P2**. no external dependencies means we can linearly scale

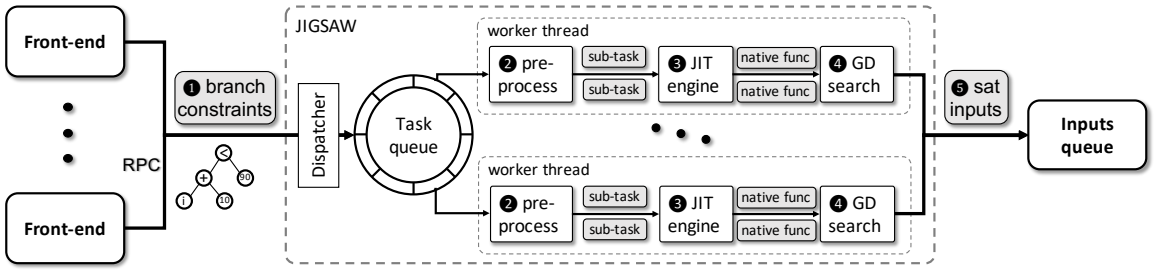

Figure 4.1: Overview of JIGSAW

the search to multiple cores without worrying about data races and lock contention. **P3**. being a function, we can easily pass the new test inputs as argument via registers or memory thus avoid going through file systems. These properties alone already eliminate two major scalability bottlenecks identified in [74], namely `fork()` and file system.

Moreover, being a straight-line function means the function does not have any conditional branches, which means **P4**. it is easier for modern processors to exploit instruction-level parallelism without worrying about branch mis-prediction during speculative execution.

Finally, since each branch predicate is much simpler than the original program under test, so **P5** fuzzing individual branch's path constraints can be orders of magnitude faster than fuzzing the whole program under test.

4.1.2 Overview

Figure 4.1 shows the design of JIGSAW. JIGSAW works in a way similar to SMT solvers: ❶ it takes a branch's path constraints (with dependencies) in the abstracted syntax tree (AST) form as input; ❷ it preprocesses the AST to find all the input bytes and constants, and decomposes it into potential sub-tasks; ❸ it then compiles each sub-task into a function in LLVM IR and uses LLVM's JIT engine to compile the IR into a native function;

❹ it searches for a satisfying solution using gradient-guided search; and ❺ if a solution is found within a time budget, it returns the solution.

```
bool test(i) { return i < 13; }
void main() {
  unsigned int x;
  read(0, &x, sizeof(x));
  if (x > 8 && test(x))
    assert(0);
}
```

Listing 1: A running example to demonstrate the work flow.

A Running Example. To demonstrate how JIGSAW works, we will use the following branch constraints as an example. In this simple example, `x` is from `stdin` and will affect the conditional branch at line 3. In step ❶, we use a symbolic execution engine to collect the path constraints. This can be done by marking the input from the `read` system call (*i.e.*, `x`) as symbolic. When the execution reaches line 5, we have a conditional branch whose branch predicate is symbolic, which can be represented as following.

```
land(
  ugt(read(0, 32), constant(8, 32)),
  ult(read(0, 32), constant(13, 32))
)
```

Note that path constraints are essentially a dynamic slice of the execution trace of the PUT, so even though there is a function call (invoking `test`) in the original code, a symbolic executor will "enter" the function and collect/slice instructions that are related to the branch, instead of collecting `test(read(0,32))`. This example also shows how dependency-based nested branch collection works. Here, both the branch on line 5 and line 1 depend on `x`, so we need to solve them together.

Next, in step ❷, we break down the (conjunct) path constraints into two sub-tasks that should be solved jointly. We will also normalize ASTs and map leaf nodes of ASTs (*i.e.*, `x` and `constant`) as arguments. This step ensures every JIT'ed native function can return a numeric distance, so we can calculate their approximated gradient to guide the search; otherwise, we can only observe two binary value: `true` and `false`. The result is as follows.

```
ugt(arg0, arg1)
ult(arg0, arg1)
```

After preprocessing, in step ❸, we compile each sub-task into a function in LLVM IR (2). Note that these functions are generated in memory using LLVM's C++ API instead of writing to files, like in [40, 49]. Following is an example (dumped as LLVM disassembly). For the sake of space, we only show one of the sub-task `ugt(arg0, arg1)`. Note that line 9 to line 13 is for calculating the distance for the gradient-guided search.

```
define i64 @rgdjit0(i64*) {
entry:
  %1 = getelementptr i64, i64* %0, i32 0
  %2 = load i64, i64* %1
  %3 = trunc i64 %2 to i32
  %4 = getelementptr i64, i64* %0, i32 1
  %5 = load i64, i64* %4
  %6 = trunc i64 %5 to i32
  %7 = zext i32 %3 to i64
  %8 = zext i32 %6 to i64
  %9 = icmp ugt i64 %7, %8
  %10 = sub i64 %7, %8
  %11 = select i1 %9, i64 zeroinitializer, i64 %10
  ret i64 %11
}
```

Listing 2: JIT'ed LLVM IR of the branch constraints in Listing 1.

After generating the IR function, we use LLVM's JIT engine to emit a native function. Note that we *do not* enable any optimizations during JIT compilation, for two

reasons: (1) path constraints are collected from already optimized code and are usually not too complex, but (2) more importantly, we found that the extra time spent on optimization will actually reduce the overall branch flipping rate because compilation is much more expensive than fuzzing (see section 4.4 for more details). In step ❹, we plug two JIT'ed functions into the gradient-guided search algorithm from Matryoshka [17] to search for a satisfying `x`. This algorithm is able to jointly solve conjunctions of sub-tasks.

4.1.3 Challenges

While directly fuzzing branch constraints is promising, we need to address two critical road-blockers.

Constraint Collection. First, in order to compile path constraints into native functions, we need to collect the constraints. Traditionally, this was done by *interpreting* each executed instruction [66, 11]. As a result, the collection phase was extremely slow. For example, Yun *et al.* [77] reported that KLEE [11] is around 3,000 times slower than native execution and Angr [66] is more than 321,000 times slower. So it could completely eclipse the benefit of faster branch solving speed. As a result, their overall efficiency is worse than the random-mutation-based fuzzing. To further reduce the overhead of constraint collection, we have implemented a symbolic tracer based on the dataflow-sanitizer [68]. Our evaluation result shows that our approach is significantly faster than existing ones, including the state-of-the-art SYMCC [53].

Constraint Compilation. While searching with JIT'ed native functions offers high throughput, a slow compilation process can become a bottleneck and cancel the benefit of faster solving speed (see section 4.4). Our solution to this problem is to cache the JIT-

compiled functions so we can avoid repeatedly *compiling* the same constraints. However, simply caching the raw JIT'ed path constraints yields a mediocre cache hit rate. The reason is that we do not see identical constraints very often. Our insight to solve this problem is that many constraints operate on different data (*e.g.*, `x > 8` and `y > 16`) are performing the same check (*e.g.*, `ugt(arg0,arg1)`); therefore we can use the same JIT'ed function to solve both constraints. Note that our function cache is different from the constraint cache used by symbolic executors. A constraint cache memorizes satisfying solutions to avoid solving the same constraints repeatedly; our function cache saves JIT'ed functions to avoid *compilation*, not solving. So, they are complementary and can be used together.

Lock Contention. While invoking JIT'ed path constraints is highly parallelizable, data races can happen in other steps (*e.g.*, updating the native function cache). A standard way to avoid data races is to use locks; however, lock contention can also scalability bottleneck. We apply two main strategies to avoid lock contention: (1) we reduce data sharing thus the locations where data race can happen; and (2) we reduce the use of locks by using lock-free data structures.

4.1.4 Comparison with SMT Solvers

Since our prototype JIGSAW is a path constraint solver, a natural question is: how it compares to SMT solvers. We believe the comparison can be done at two levels. Methodology-wise, our approach provides a new and fast way to evaluate the satisfiability of a concrete model (*i.e.*, assignments to symbolic variables); therefore, our approach can also be leveraged by SMT solvers to improve their performance. For example, we have used path constraints collected from `objdump` to evaluate the `z3_model_eval()` API and JIT'ed

functions from JIGSAW: the Z3 API can evaluate about 43K concrete models per second while JIGSAW can evaluate 8M models per second.

At the tool level, our prototype JIGSAW has both advantages and limitations. First, due to the high search throughput, our evaluation shows that JIGSAW can solve path constraints faster than SMT solvers. However, because JIGSAW only employs a single search heuristic (the gradient-guided search from [16]), it is not as capable as off-the-shelf SMT solvers. First of all, JIGSAW can only be used to find satisfying inputs, while SMT solvers can also be used to prove theorems. Second, our current prototype only supports constraints in the theory of bit-vectors while most modern SMT solvers support more theories like arrays, floating-point numbers, and strings. Even for bit-vectors, JIGSAW cannot identify unsatisfiable constraints and can only solve 94% of the constraints solved by Z3. Nevertheless, we want to emphasize that these limitations are introduced by the search heuristic but not the methodology proposed in this work. Therefore, these limitations can be addressed by adopting additional heuristics from SMT solvers. For instance, similar to Bitwuzla [45], we can apply rewriting rules to identify simple unsatisfiable constraints and add a bit-blasting-based solver to handle constraints that cannot be decided by local search.

4.2 Design

In this section, we present the design details of JIGSAW.

```
message AstNode {
  uint32 op;
  uint32 width;      // operand width
  string value;      // used by constant expr
  string name;       // used for debugging
  uint32 offset;     // used by read expr
  uint32 label;      // for expression dedup
  uint32 hash;       // for request dedup
  repeated AstNode children;
}
```

Listing 3: AST node for function cache lookup.

4.2.1 Getting Constraints

JIGSAW relies on a concolic execution engine to collect path constraints to be solved. To do so, we use our data-flow sanitizer-based engine (section 4.3). Similar to SymCC [53], our engine collects path constraints at the LLVM IR [55] level. The collected path constraints are then passed to JIGSAW through shared memory.

AST for Cache Lookup. 3 shows the format of each abstract syntax tree node we use to store the collected constraints, where `op` denotes the operator and `children` denote the child nodes of the AST. Currently, JIGSAW supports all of LLVM's binary operators, including integer arithmetic, bitwise, and logical instructions. It also supports three conversion operators (`ZExt`, `SExt`, and `Trunc`) and relational comparison instructions. We add a special operator `Read` to denote symbolic input bytes. Different input bytes are distinguished with their offset from the beginning of the input.

Nested Branches. One particular challenge during branch flipping is that solving a single branch predicate alone is not enough [17]. The reason is that the solution can negatively affect preceding branches and cause the control-flow to diverge; as a result, the new input

may never reach this supposedly solved branch. To address this problem, we need to solve these dependent/nested branches together. In this work, we used QSYM's [77] approach to identify nested branches based on data dependency: finding all precedent branches whose input bytes overlap with the current branch.

4.2.2 Preprocessing

Since calculating the numeric approximation of gradient works best for individual comparison instructions where we can measure the distance, we want to avoid logical operators inside the JIT'ed testing function. Therefore, after receiving a solving request, the first step is to break it up into possible sub-tasks, where each sub-task is a single AST rooted with a comparison instruction. Then we will parse the AST to find all the arguments (both input bytes and constants) to the testing function.

Removing Logical Or. Due to compiler optimizations, branch constraints may occasionally contain logical or (`LOr`) operators. To remove `LOr` operators, we first convert a solving request into DNF (disjunctive normal form). Each clause in the DNF can then be solved in parallel. As long as one clause is solved, the branch can be flipped.

Removing Logical And. After removing `LOr`, each sub-task should be clauses connected with logical and (`LAnd`). To remove `LAnd`, we will generate a separate testing function for each clause. However, all clauses will be solved jointly (subsection 4.2.4).

Removing Logical Not. After removing `LOr` and `LAnd`, we may still have clauses/AST with a leading logical not (`LNot`) operator. Removing `LNot` is relatively simple, we just remove it and set the comparison condition to its opposite (*e.g.*, $<$ to $\geq$).

Table 4.1: Transforming a comparison operation into a distance-based loss function.

Comparison	Loss function $f()$
$slt(a, b)$	$max(sext(a, 64) - sext(b, 64) + \epsilon, 0)$
$sle(a, b)$	$max(sext(a, 64) - sext(b, 64), 0)$
$sgt(a, b)$	$max(sext(b, 64) - sext(a, 64) + \epsilon, 0)$
$sge(a, b)$	$max(sext(b, 64) - sext(a, 64), 0)$
$ult(a, b)$	$max(zext(a, 64) - zext(b, 64) + \epsilon, 0)$
$ule(a, b)$	$max(zext(a, 64) - zext(b, 64), 0)$
$ugt(a, b)$	$max(zext(b, 64) - zext(a, 64) + \epsilon, 0)$
$uge(a, b)$	$max(zext(b, 64) - zext(a, 64), 0)$
$a = b$	$abs(zext(a, 64) - zext(b, 64))$
$a \neq b$	$max(-abs(zext(a, 64) - zext(b, 64)) + \epsilon, 0)$

Arguments Mapping. To maximize the reuse of JIT'ed functions and minimize the compilation time (subsection 4.2.5), we treat both input data and constants as arguments to the testing function. In our current design, the testing function takes a single argument as an array of 64-bit integers, to support an arbitrary length of arguments. To correctly invoke the testing function, we need to map input bytes and constants in the AST to the correct offsets inside the argument array. To do so, we perform a pre-order traversal of the AST and number the leaf nodes according to the traversal order.

4.2.3 Code Generation

After preprocessing a solving task and decomposing it into sub-tasks, the next step is to JIT-compile each comparison AST into a testing function that returns a distance so we can perform a gradient-guided search. To do so, we transform the comparison instruction into a loss function similar to previous works [16, 17, 65]. Table 4.1 shows the transformation. To minimize the impact of integer overflow/underflow during calculation, we first extend both operands into 64-bit numbers. For each unsigned comparison, we perform a zero extension

(`ZExt`). For each signed comparison, we perform a signed extension (`SExt`). Then we apply the `max` operation to avoid any negative distance. This is done by performing the original comparison followed by a conditional move (*i.e.*, `Select`) instruction. Because our AST language is close to LLVM IR, the rest of the code generation is straightforward: just perform a post-order traversal of the AST.

4.2.4 Solving

To search for a satisfying input, we use the gradient-guided search algorithm from Matryoshka [17], which uses a numeric approximation to calculate the gradient and is capable of solving conjunctions of comparisons. The original algorithm uses three search strategies to solve conjunctions of branch constraints, in our prototype, we used a simplified version:

1. Prioritize satisfiability: try to solve the current branch predicate first.

2. Once we find a satisfying input, use the following loss function to solve nested branch constraints using joint optimization:

$$g(\mathbf{x}) = \sum_{i=1}^{n} f_i(\mathbf{x})$$

To avoid negating a previously satisfied constraint, we will stop mutating an input byte if its new value will violate any constraint that is satisfied previously. However, as long as the constraint is satisfied, we will allow the input byte to be mutated according to the gradient.

Handling Division and Remainder. During fuzzing, the JIT'ed function may generate divide-by-zero exceptions. Instead of capturing and recovering from such exceptions, we

add a check before each divide instruction to see whether the divisor is zero and if so, we simply skip the execution of the current input. Note that this handling will not prevent JIGSAW from finding a satisfying solution, as a solution that will trigger a divide-by-zero exception is not a satisfying solution. This handling will not prevent the coverage-guided testing from discovering divide-by-zero bugs either. To detect divide-by-zero bugs in the PUT, the concolic executor needs to explicitly check if divide-by-zero is possible (*i.e.*, adding an assertion for $divisor \neq 0$) under the current path constraints.

4.2.5 Scaling

While the single thread design presented so far already provides a much higher branch flipping rate than existing fuzzers (*e.g.*, AFL and Angora), another major design goal of JIGSAW is to provide linear scalability to multiple cores. As mentioned in section 4.1, searching for a satisfying input with JIT'ed path constraints should be highly scalable, as there are no interdependencies between different solving threads. However, constructing the solving task may become a bottleneck. In this subsection, we discuss how we improve the scalability of task construction.

Parallelized Solving. We scale the solving to multiple cores using threads instead of processes, as communication through shared memory is easier and more efficient. Moreover, two properties of our JIT'ed functions allow us to do so. (1) They have no side effects after the invocation, so we do not need to clean up. (2) They do not have external dependencies, so we do not need to worry about interference between different threads. Finally, thanks to property (1), we can further avoid the expense of creating threads by using a thread pool,

because once a side-effect-free solving task is done, the thread is ready to handle another task.

Function Cache. While LLVM's JIT engine is easy to adopt, it is also much slower than other JIT engines like the TCG (tiny code generator) from QEMU. Fortunately, many path constraints collected during fuzzing are very similar (*e.g.*, performing the same check over different input data). Based on this observation, we designed a function cache to minimize the invocation of the JIT engine. To further maximize the reuse of compiled testing functions, we also treat all constants in the constraints as input arguments to the testing function. By doing so, constraints like $a + b < 10$ and $c + d < 40$ can now reuse the same testing function.

Essentially, our function cache maps a partial AST (*excluding all leaf nodes*) to a compiled function. To speed up the look-up and tree comparison, we added a *hash* value to each AST node. Since we treat both input bytes and constants as arguments to the testing function, each leaf node has a hash value according to the preorder traversal (*i.e.*, the corresponding argument index). For each non-leaf node, its hash is calculated using its operator and the hash(es) of its operand(s) (*i.e.*, hash(es) of child node(s)). This is similar to a Merkle tree (except we do not use a crypto hash function), so if the hash values of two ASTs are different, we do not need to perform a more expensive recursive equality comparison.

Since it is important to maintain a high cache hit rate, we use a global function cache instead of per-thread caches.

Avoiding Lock Contentions. We minimize the use of locks. First, each task construction thread has its own LLVM JIT engine to avoid sharing. Second, the dispatcher and solving threads communicate with a lock-free queue. Third, we implement the function cache

with a lock-free hash table. Finally, we minimize dynamic memory allocation and use the TCMalloc [32] from Google to reduce contentions caused by `malloc` and `free`.

4.3 Implementation

In this section, we provide some implementation details of JIGSAW and additional components to support end-to-end fuzzing.

JIGSAW. We implemented JIGSAW in C++ with about 4,800 lines of code. The gradient-guided search algorithm is a re-implementation of Angora's. We used the ORC JIT APIs from LLVM for JIT compilation. We used the CTPL[1] for the thread pool, and an open-source implementation based on linear probing[2] for the hash table. For the heap allocator, we used the TCMalloc from Google.

Constraint Collector. JIGSAW can support different symbolic executors as the front-end constraint collector. In our evaluation, we used our concolic execution engine based on the data-flow sanitizer (DFSan)[3]. We chose this DFSan-based constraints collector for a better comparison with Angora [16]. We re-implemented QSYM's dependency forest [77] to identify nested branches. To support C++ programs, we used the instrumented `libc++` library.

Hybrid Fuzzer. JIGSAW itself acts as a solver. To perform end-to-end coverage-guided test generation, we still need a fuzzing driver to close the loop. For the evaluation, we implemented a hybrid fuzzer based on Angora [16].

[1] `https://github.com/vit-vit/ctpl`
[2] `https://github.com/cmuparlay/parlaylib/`
[3] `https://github.com/ChengyuSong/Kirenenko`

4.4 Evaluation

In this section, we evaluate our prototype JIGSAW, aiming to answer the following research questions.

- **RQ1**: Does it improve the search throughput?
- **RQ2**: Can it improve the branch flipping rate?
- **RQ3**: Can it scale well with the increase of CPU cores?
- **RQ4**: Can it improve the performance of coverage-guide testing?

Experiment Setup. All evaluation was done on a workstation with two-socket, 48-core, 96-thread Intel Xeon Platinum 8168 processors. The workstation has 768G memory. The GPU is Quadro P5000. To minimize the impact of I/O, we used four Intel 512G Pro 7600 NVME SSD in a RAID-1 setup. The operating system is Ubuntu 18.04 with kernel 5.4.0. The file system is XFS. JIGSAW was compiled with LLVM 9.0.0 with `-O3`. For Z3, we used version 4.8.7.

Dataset. We used two datasets in our evaluation. The first dataset includes 14 real-world programs (Table 4.2). We use this dataset to answer RQ1, RQ2, and RQ3. To answer RQ4, our main dataset is the Google Fuzzbench [34]. To compare with fuzzers that are not supported by Fuzzbench, we use part of our first dataset.

4.4.1 Constraint Solving Performance

To evaluate the solving performance, we collected about 10 million path constraints from 14 real-world programs (Table 4.2). We first use AFL to fuzz the target programs for 48

Table 4.2: Details of real-world applications used for evaluation.

Program	Version	#Constraints	Program	Version	#Constraints
objdump	2.33.1	372,880	libpng	1.2.56	626,480
size	2.33.1	604,610	openssl-x509	b0593c0	1,000,000
nm	2.33.1	1,000,000	libjpeg-turbo	b0971e4	494,695
readelf	2.33.1	1,000,000	mbedtls	4c08dd4	377,542
tiff2pdf	4.1.0	803,036	libxml2	2.9.2	942,240
file	5.39	1,000,000	vorbis	c1c2831	47,387
tcpdump	4.9.3	1,000,000	sqlite3	c78cbf2	769,548

Table 4.3: Solving capability comparison.

Solver	Nested	vs. Z3-60s	Single	vs. Z3-60s
Z3-60s	50.07%	-	89.17%	-
STP	49.04%	0.98	89.13%	1.00
YICES2	49.07%	0.98	89.05%	1.00
Bitwuzla-60s	50.17%	1.00	89.13%	1.00
Bitwuzla-LS-1M	48.36%	0.97	88.40%	0.99
JIGSAW-1M	46.96%	0.94	87.97%	0.99

hours (single instance, non-deterministic mode, no dictionary). Then we ran our DFSan-base constraint collector over the corpora generated by AFL and serialized path constraints required to negate every branch to files. We chose to load the collected constraints from files to minimize the impact of the constraint collector (which will be evaluated in subsection 4.4.2). Because the numbers of seeds found by AFL vary a lot across the programs, to ensure we have enough constraints from every program, we only applied a light filter when collecting the constraints, which avoids duplicated constraints from the same seed. For programs with more seeds, we cut off at 1 million. The collected path constraints include both satisfiable and unsatisfiable ones, reflecting the real scenario during hybrid fuzzing.

Solving Capability. Before evaluating the search throughput and branch flipping rate, we first compared JIGSAW's solving capability with a set of SMT solvers that provide

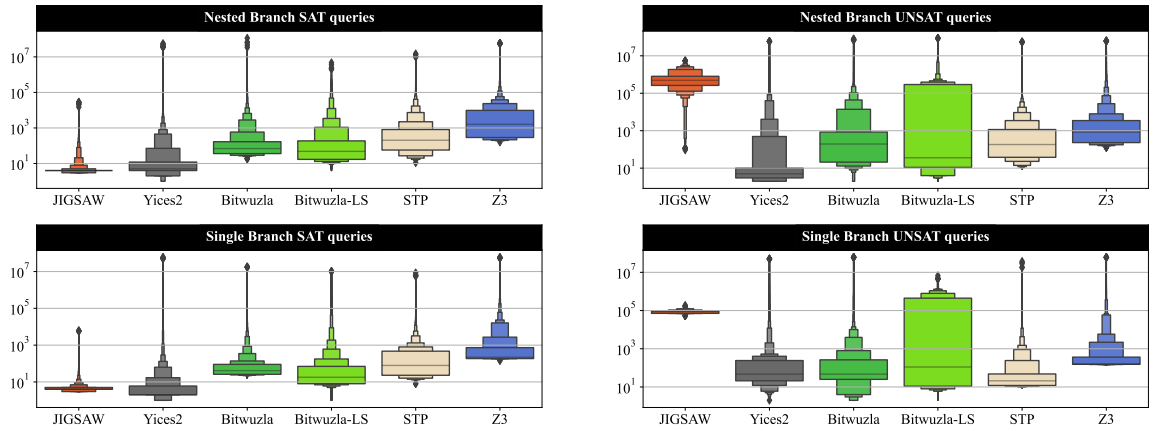

Figure 4.2: Constraints processing time distribution (in micro-seconds).

C/C++ bindings, including Z3 [21], STP [27], Yices2 [22], and Bitwuzla [45]. For Bitwuzla, we evaluated two different modes: (1) the configuration that won the SMT-COMP 2021 [46] (denoted as Bitwuzla), and (2) the configuration that only uses local search without bit-blasting (denoted as Bitwuzla-LS). Besides trying to understand the limitations of the gradient-guided search heuristic, this evaluation also helps us to set the proper timeout for the following experiments. For this purpose, we used large timeout setups in this experiment: 1 million iterations for JIGSAW (denoted as JIGSAW-1M) and Bitwuzla-LS, and 60 seconds for Z3 and Bitwuzla. For STP and Yices2, we either did not find a timeout setting or the timeout functionality did not work well, to avoid getting stuck in the middle of the evaluation, we removed timeout constraints (using Z3) from the dataset when evaluating these two tools. Note that the version of Z3 we used (4.8.7) does not support timeout on `get_model()`, the API to retrieve a satisfying assignment. So we modified its source code to support a timeout. As a result, there are cases where Z3 deems the constraints are satisfiable but cannot return a model within the timeout. We consider these cases as *not solved*.

The result is shown in Table 4.3. All the results returned by JIGSAW were verified by Z3 to validate their correctness. At 1M iterations, JIGSAW was able to solve 93.8% of the nested branch constraints Z3 can solve within 60 seconds. We also evaluated last-branch constraints because, in QSYM [77], the authors have demonstrated that inputs satisfying just the last branch can also lead to new coverage in many cases. For last-branch constraints (*i.e.*, without nested dependencies), JIGSAW was able to solve 98.65% of the constraints Z3 can solve within 60 seconds. Based on the results, we conclude that JIGSAW's simple gradient-guided search algorithm (subsection 4.2.4) is capable enough to solve most constraints, especially last branch constraints.

To understand why certain constraints are not solved by JIGSAW, we analyzed the distribution of the following factors in the solved and unsolved constraints: (1) involved operations, (2) AST size of a constraint, and (3) the number of nested constraints. The result shows the two most important factors. First, a large portion of constraints with `udiv`, `urem`, and `xor` are not solved by JIGSAW, due to the loss of gradient. Specifically, when estimating the gradient, the algorithm adds a small ϵ (± 1) to each input byte and then calculates the change of distance to the objective (Table 4.1). However, when the constraints include division or bitwise masking, ± 1 is usually too small to change the distance, so the gradient estimation would fail. The second factor is a well-known limitation of gradient-guided search: when the constraints are not convex, the joint-optimization can get stuck at a local minimum. On the contrary, the backtracing strategy used by SMT solvers can avoid this. We want to emphasize again that these are the limitations of the search heuristic used in our prototype, but are not limitations of the proposed methodology (*i.e.*, using JIT'ed path constraints to

evaluate inputs); and our approach can be combined with other search heuristics to overcome these limitations.

Solving Efficiency. Figure 4.2 shows the solving time distribution. For JIGSAW, solving time is the fuzzing time. For SMT solvers, solving time includes checking for satisfiability and retrieving the solution/model. As we can see, for satisfiable (sat) constraints, JIGSAW is faster than all but Yices2. The biggest difference between JIGSAW and other solvers is for unsolvable (unsat) constraints. Because JIGSAW cannot tell if a set of constraints are not satisfiable, it can only timeout. As a result, unsat constraints will consume a lot of time if we set JIGSAW's timeout to a large number of iterations. On the contrary, SMT solvers can tell whether a set of constraints are unsat rather quickly. We also analyzed the most important factors that would affect JIGSAW's solving time using linear regression. As expected, the top ones are the size of the constraint's AST, the number of nested constraints, and the presence of division operations. We would like to point out again that lacking the ability to answer unsat queries is not a fundamental limitation of our methodology, but a limitation of our current prototype; and it can be addressed by incorporating more rewriting rules and a bit-blasting solver similar to Bitwuzla.

Choosing Timeout Setups. To enable more fair comparisons between different tools on the metric of branch flipping rate, we need to select appropriate timeout setups. Specifically, the branch flipping rate is calculated as:

$$\text{branch flipping rate} = \frac{\text{number of satisfying solutions}}{\text{total process time}}$$

Therefore, (1) a too-short timeout will reduce both the numerator (number of satisfying solutions) and the denominator (total processing time), and (2) a too-long timeout will

unnecessarily increase the denominator. To address this issue, we can either fix the numerator or fix the denominator. In the following experiments, we decided to fix the numerator because we do not know the distribution of easy, hard, and unsat constraints in the dataset, so if different solvers are not solving the same set of constraints, then the results could be biased. To this end, we leveraged the experimental results in Figure 4.2 to determine the timeout setups. Specifically, we set the timeout for JIGSAW at 1,000 iterations (denoted as JIGSAW-1K), which can solve 93.8% of all the constraints that Z3 can solve within 60 seconds. Similarly, we set the timeout at 50ms for Z3 (denoted as Z3-50ms), at 6ms for Bitwuzla (denoted as Bitwuzla-6ms), and at 10,000 model updates for Bitwuzla-LS (denoted as Bitwuzla-LS-100K), which can solve 94.0%, 92.5%, and 94.5% of Z3-60s.

Table 4.4: The throughput (number of tried inputs per second) of JIGSAW (JIGSAW-1K) and Bitwuzla (BZLA-LS-100K) in a single-threaded execution.

Program	Nested Branch Constraints		Last Branch Constraints	
	JIGSAW	BZLA-LS	JIGSAW	BZLA-LS
objdump	382.3 (±2.1)K	25.7 (±0.5)K	4.3 (±0.8)M	84.9 (±1.4)K
size	1995.5 (±31.2)K	42.3 (±0.5)K	6.7 (±0.8)M	67.1 (±0.3)K
nm	4649.5 (±33.6)K	45.0 (±1.0)K	13.4 (±1.1)M	82.6 (±0.9)K
readelf	622.7 (±8.9)K	28.0 (±0.0)K	7.2 (±0.5)M	90.2 (±0.1)K
libpng	820.0 (±5.1)K	28.1 (±0.0)K	1.2 (±0.3)M	121.3 (±1.0)K
tiff2pdf	280.5 (±7.9)K	29.2 (±0.5)K	4.5 (±0.4)M	82.4 (±1.0)K
file	431.7 (±6.2)K	40.0 (±0.9)K	4.9 (±0.2)M	56.3 (±0.8)K
tcpdump	1396.9 (±18.4)K	41.3 (±0.9)K	1.7 (±0.1)M	82.0 (±0.9)K
openssl	270.2 (±40.4)K	57.4 (±0.1)K	4.2 (±0.1)M	102.5 (±0.5)K
sqlite3	3446.4 (±51.8)K	46.5 (±1.5)K	0.8 (±0.0)M	54.3 (±0.0)K
vorbis	358.4 (±11.5)K	38.5 (±0.3)K	3.1 (±0.0)M	101.0 (±1.8)K
mbedtls	61.1 (±1.6)K	37.3 (±0.8)K	2.8 (±0.1)M	26.2 (±0.2)K
libxml2	2629.4 (±13.5)K	21.0 (±0.0)K	2.4 (±0.1)M	69.1 (±0.0)K
libjpeg-turbo	110.6 (±4.1)K	6.9 (±0.0)K	0.9 (±3.5)M	42.9 (±0.1)K
Geomean	637.2K	31.7K	3.1M	71.2K

Table 4.5: The branch flipping rate of single thread JIGSAW and comparison with popular SMT solvers.

Program	Nested Branch Constraints					Single Branch Constraints				
	Yices2	Boolector	STP	Z3	JIGSAW	Yices2	Boolector	STP	Z3	JIGSAW
objdump	100.9	8.1	37.1	22.5	193.3	21.9 K	0.9 K	2.6 K	0.5 K	44.4 K
size	378.0	64.4	126.5	54.4	1263.0	13.9 K	0.8 K	1.0 K	0.6 K	23.7 K
nm	4109.0	539.7	777.3	294.5	4709.0	39.7 K	0.4 K	4.9 K	0.5 K	21.2 K
readelf	319.5	128.9	171.9	42.9	371.2	11.5 K	0.1 K	1.1 K	1.0 K	41.1 K
libpng	714.9	54.7	268.7	120.3	505.3	14.4 K	0.6 K	1.1 K	0.1 K	50.3 K
tiff2pdf	174.9	31.1	40.9	24.6	133.6	40.5 K	0.5 K	12.0 K	1.2 K	96.7 K
file	80.7	26.9	46.5	14.0	187.3	22.4 K	1.9 K	2.9 K	0.4 K	25.6 K
tcpdump	1095.9	139.2	437.7	92.5	599.1	23.7 K	1.1 K	4.5 K	0.5 K	7.4 K
openssl	2.4	1.4	27.8	21.7	186.0	6.6	10.8	0.6 K	0.5 K	6.7 K
sqlite3	21908.5	2228.4	2788.1	896.5	9886.4	70.2 K	6.9 K	10.5 K	2.0 K	100.0 K
vorbis	70.7	31.0	35.3	7.0	114.4	1.2 K	1.4 K	1.6 K	0.2 K	0.8 K
mbedtls	31.1	4.5	11.5	5.3	98.4	4.5 K	0.3 K	0.3 K	16.3	3.5 K
libxml2	2825.8	292.0	553.1	191.6	2556.2	44.0 K	4.1 K	6.4 K	0.3 K	54.7 K
libjpeg-turbo	2.1	0.5	2.9	4.7	23.1	92.4	8.0	208.2	53.7	362.0
Geomean	212.4	38.3	97.6	41.0	411.2	6.8 K	0.5 K	1.9 K	0.3 K	15.1 K

Single-thread Search Throughput. Because our primary design goal is to improve the search throughput, we first evaluated JIGSAW's throughput and compared it with Bitwuzla's local search mode. Similar to the previous experiment, all constraints were first loaded into memory, then passed to the solver one by one. For each set of constraints, we ran the corresponding experiments 30 times and report the average and standard deviation. Table 4.4 shows the result. The first half is for nested branch constraints and the second half is for the last branch constraints. On nested branch constraints, our search throughput ranges from 61.1K to 4.6M inputs/sec with a single thread. For last branch constraints, as fewer functions need to be evaluated, our search throughput is much higher, ranging from 755.3K to 13.4M inputs/sec. The throughput on `nm` is much higher than others because only about 25% of the collected constraints are solvable; so JIGSAW spent more time searching for a result with the JIT'ed functions. To put these numbers into context, Angora's search throughput ranges from 58 (`file`) to 3363 (`libpng`) inputs/sec on the same machine. On average (geomean

Table 4.6: Accumulated solving time breakdown of JIGSAW

Preprocessing	JIT	Searching	Cache Hit Rate
1328s	462s	4403s	99.99%

Table 4.7: Benefits of using function cache, when solving 20,000 constraints from `readelf`.

Caching	Hit Rate	JIT	Searching	Throughput
Disabled	N/A	33.9s	12.6s	229K inputs/s
Full AST	66.9%	12.1s	12.6s	394K inputs/s
Normalized AST	99.9%	0.7s	12.6s	747K inputs/s

across all programs) JIGSAW's throughput (on nested branch constraints) is about *two orders* of magnitude higher (373×). Compared to Bitwuzla, JIGSAW's throughput is also much higher. Based on this experiment, we believe **the answer to RQ1 is yes**: our approach indeed can significantly improve the search throughput.

Single-thread Branch Flipping Rate. Next, we compared our branch flipping rate with popular SMT solvers. As shown in Table 4.5, JIGSAW's branch flipping rate is also very good when compared to SMT solvers: JIGSAW can beat other solvers on branch flipping rate, including Yices2 and Bitwuzla (because of the shorter timeout setup). On average, JIGSAW is 14.4× faster than Z3 on solving nested branch constraints and 119.7× faster on solving single branch constraints. Based on this comparison, we believe **the answer to RQ2 is yes**: when the search throughput is high enough, even with a simple search heuristic, JIGSAW can flip branches faster than state-of-the-art tools.

Solving Time Breakdown. Table 4.6 shows the accumulated time spent on different components of JIGSAW (preprocessing, JIT, and fuzzing), and the average function cache hit rate. The timeout is at 1K iterations. As we can see, even with a high cache hit rate

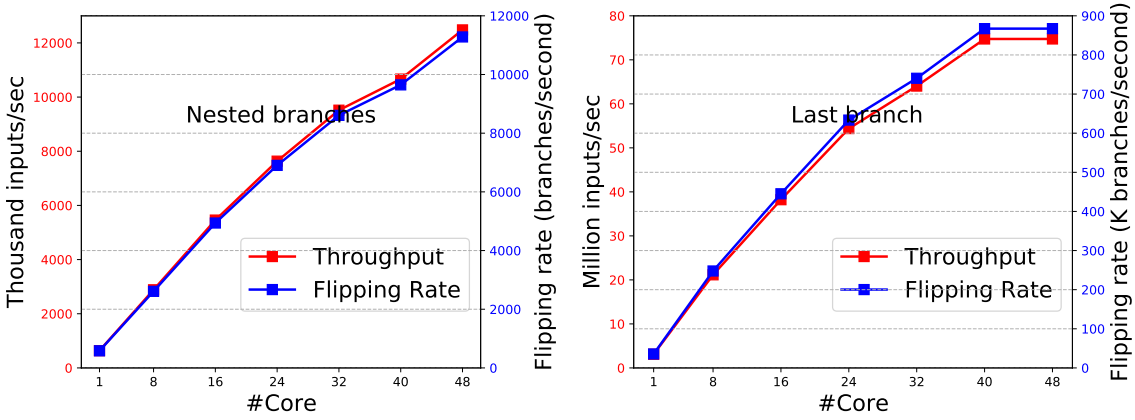

Figure 4.3: Average search throughput and branch flipping rate of JIGSAW on multiple cores.

(99.99%), a significant portion of time is still spent on JIT compilation. Therefore, the performance could be worse if without the code cache or with an even slower JIT procedure (*e.g.*, that used by [58, 49]). Similarly, we can further improve the performance by using a faster JIT engine and by making the cache persistent.

Effectiveness of Function Cache. To better understand the impact of our normalized AST to function cache, we did a comparison using 20K constraints from `readelf`. The result is shown in Table 4.7. As we can see, when we enable cache with full AST matching, the cache hit rate is only 66.9%, the JIT time is reduced by 64.3%, and the throughput is mildly increased by 72.1%, compared to no function cache. With our optimization that normalizes the AST before matching, the cache hit rate increases significantly to 99.9%. The corresponding JIT time is reduced by 97.9%, and the throughput is increased by 3.3× compared to no function cache.

Multi-thread Performance. In this subsection, we evaluate JIGSAW's scalability to multiple cores. We focus on two main performance metrics: search throughput and branch

flipping rate. We tested with 8-, 16-, 24-, 32-, 40-, and 48-threads, each thread is pinned to a real CPU core (not hyper-thread). For comparison, we also tried multi-threaded Z3 where each thread uses a separate Z3 context and solver.

Figure 4.3 shows the results. Overall, adding more threads/cores can help JIGSAW increase the throughput and branch flipping rate. The geomean of JIGSAW's throughput can reach 12.5M inputs/sec for solving nested branch constraints and 74.7M inputs/sec for solving single branch constraints. The geomean of JIGSAW's branch flipping rate can reach 11.3K branches/sec and 860.0K branches/sec, respectively. For Z3, we did not observe much improvement when adding more parallelism, due to lock contention. Based on this experiment, we conclude that the **answer to RQ3 is yes**: our approach can scale well to multiple cores.

To put the numbers into context, Xu *et al.* reported a throughput of around 6.5M inputs/sec when fuzzing `libpng` with libFuzzer, using 120 CPU cores and their new OS primitives [74]. For `libpng`, the peak throughput of JIGSAW, using 48 cores, can reach 18.1M inputs/sec for solving nested branch constraints and 36.9M inputs/sec for solving single branch constraints; which is about 22.1× and 30.9× faster than single thread mode, respectively. The corresponding branch flipping rate can reach 15.0K branches/sec for solving nested branch constraints and 895.1K branches/sec for solving single branch constraints; which is also about 21.4× and 31.0× faster than single thread mode, respectively.

Table 4.8: Comparison of concolic execution engines on flipping all symbolic branches along a single execution trace.

Programs	JIGSAW	Z3-10s	Z3-50ms	Angora	SymCC	Fuzzolic
readelf	2.2h	51.3h	12.6h	89.5h	546.6h	48.2h
objdump	12.3h	227.5h	29.6h	411.5h	373.5h	52.2h
nm	0.3h	18.1h	3.2h	72.3h	29.3h	48.2h
size	0.1h	8.4h	1.4h	16.8h	12.6h	5.2h
libxml2	0.2h	9.3h	3.6h	58.0h	52.3h	20.9h
readelf	7923	7957	7423	8287	6410	5843
objdump	4926	4926	4865	4846	4929	4689
nm	3347	3347	3329	3339	3122	3123
size	2453	2457	2449	2406	2229	2259
libxml2	6038	6233	6034	5952	6012	6022

4.4.2 End-to-End Fuzzing Performance

In this subsection, we evaluate the effectiveness of JIGSAW on coverage-guided test generation. We choose the Z3 solver as the main comparing target in the end-to-end fuzzing evaluation for the considerations below:

- Z3 is widely adopted by recently concolic executors, such as QSYM [77], SymCC [53], SymQEMU [54], and Fuzzolic [9]. Using Z3 makes it easier to tell how much performance gain is from our DFSan-based constraint collection engine, and how much is from JIGSAW.
- All other solvers are not as robust as Z3 and can get stuck in the middle of a fuzzing campaign because they either do not provide APIs to specify a timeout (Yices2) or that API does not work well (STP).

Concolic Execution Performance. We first compare JIGSAW with other state-of-the-art concolic execution (CE) engines and fuzzers on flipping all symbolic branches along execution traces of a fixed set of seeds. As argued in [54], this experiment setup removes the

path scheduling variable from the comparison so the result can better reflect the end-to-end branch flipping performance (*i.e.*, path constraints collection + constraints solving). The first two configurations to compare are Z3-10s and Z3-50ms, which share the same hybrid fuzzing driver as JIGSAW but use Z3 as the solver. The 10 seconds timeout is the setting used by other concolic executors [77, 53, 54] and 50ms timeout is the setting that offers a similar solving capability as JIGSAW-1K. The next one is Angora [16]. We believe the comparison with Angora is especially meaningful because: (1) Our constraint collector and Angora's taint analysis are both implemented based on DFSan; and (2) JIGSAW uses the same gradient-guided searching algorithm as Angora so the main difference is the search throughput. In short, JIGSAW, Z3, and Angora are almost identical except for how they try to flip a particular branch: Angora [16] performs gradient-guided search with the original program, JIGSAW performs gradient-guided search with the JIT'ed path constraints, and Z3 performs SMT solving with path constraints. We believe this setup can better reflect JIGSAW's impact on end-to-end fuzzing. We also compared with SymCC [53], a state-of-the-art CE engine also uses compile-time instrumentation to collect constraints. Since it also uses Z3 as the solver, comparison with it shows the advantages of our DFSan-based constraint collector. The last one is Fuzzolic [9], with Fuzzy-Sat [8], another fuzzing-based constraint solver. Note that we have disabled input level timeout so all tools will finish flipping all branches in one seed before moving on to the next.

Table 4.8 shows the results over the corpora from Neuzz[4]. As we can see, JIGSAW can flip branches much faster than other tools. Z3-50ms was faster than Z3-10s but also flipped fewer branches (*i.e.*, achieved lower code coverage). We want to point out that

[4] `https://github.com/Dongdongshe/neuzz`

most other tools cannot even finish processing the corpora in 24 hours, which means under a normal fuzzing setup where the seed level timeout is enabled, they may have problems flipping branches in deep execution traces.

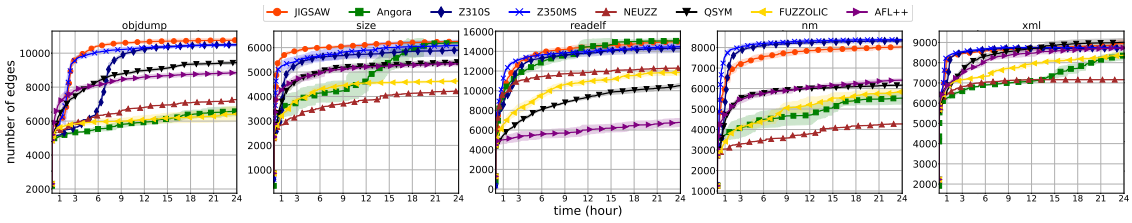

Figure 4.4: Edge coverage growth over time for local fuzzing.

Local Fuzzing. Next, we evaluated three fuzzers that are not supported by Fuzzbench. The first one is Angora [16]. We want to emphasize again that the comparison between JIGSAW, Z3, and Angora (where everything is the same except the solver) can better reflect JIGSAW's impact on end-to-end fuzzing. Note that for a better comparison with Angora, JIGSAW and Z3 use Angora's AFL mutator instead of AFL++ in this experiment. The second one is QSYM [77], a state-of-the-art hybrid fuzzer, paired with AFL++. The third one is Neuzz [64], which also uses gradient-guide search. However, instead of using numerical approximation, it uses a neural network to approximate the program under test. The fourth one is Fuzzolic [9] with Fuzzy-Sat [8] as the solver. Finally, we also included AFL++ [24] (3.12c with `cmplog` enabled), the state-of-the-art fuzzer.

For comparison, we used the same strategy as Neuzz [64]. Specifically, all fuzzers use a larger set of initial corpus instead of a single seed. To facilitate better reproducibility, we used the corpora from the Neuzz repository. We chose 5 programs from Table 4.2: `readelf`, `objdump`, `nm`, `size`, and `libxml2`, as we can find the corresponding corpus from Neuzz's

repository and they can be successfully compiled by Angora. Note that we did not use the binaries in Neuzz's repository because both JIGSAW and Angora need to compile the target program from the source code. To ensure a fair comparison, we followed Fuzzbench's setting: each fuzzing trial runs inside a docker container which is assigned and limited to one physical CPU core. The only exception is Neuzz, which also uses a dedicated GPU (P5000). All experiments are run 10 times, except Neuzz, which we cannot run in parallel. We use afl-cov to measure the edge coverage with binaries built for Neuzz.

Figure 4.4 shows the accumulated coverage growth. Compared to the two Z3 configurations, JIGSAW is better on `objdump` and `size`, worse on `nm`, and similar on `readelf` and `libxml`. This is similar to the CE testing results: given enough time (*i.e.*, the per-input timeout), Z3 can solve more constraints and achieve higher coverage; otherwise, JIGSAW can go deeper into the execution trace and flip more branches. Compared to Angora, JIGSAW's coverage growth is much faster, reflecting the advantage of its higher search throughput. For the rest fuzzers, JIGSAW is significantly better on the four binutils programs in terms of both final coverage and the coverage growth rate; it achieved similar final coverage as QSYM and AFL++ on `libxml`, but the coverage growth rate is higher.

Fuzzbench. Next, we compared JIGSAW with other popular fuzzers on Google the Fuzzbench dataset [34]. We used two configurations of our fuzzing driver. The first one uses JIGSAW as the solver, denoted as JIGSAW. The second one uses the same setup except using Z3 with 10s timeout as the solver, denoted as Z3. This setup is to show the benefit of

Table 4.9: Comparing JIGSAW with other state-of-the-art symbolic executors based on their publicly available Fuzzbench results.

Target	JIGSAW	Z3	SymCC	SymQEMU	Fuzzolic	AFL++
curl	**17956.5**	17931.0	17622.0	17564.5	17599.5	17948.5
freetype	**28026.0**	27932.5	25496.0	24028.0	26371.0	27956.5
harfbuzz	8705.0	**8959.0**	8482.5	8482.5	8515.0	8427.0
lcms	**3872.0**	2874.0	3701.5	3656.0	3770.0	3446.0
libjpeg	3809.0	3802.5	3810.5	**3819.0**	3814.0	3798.0
libpng	2128.0	2124.0	1914.5	**2149.5**	2146.5	2080.5
libxml2	13010.5	**13056.0**	11097.0	12305.0	12072.0	12429.5
libxslt	**19083.5**	19064.5	18577.0	18592.5	18515.0	18963.5
mbedtls	8297.0	**8310.0**	8260.0	8244.5	8268.0	8252.5
openssl	13768.0	13778.0	13777.0	13777.0	13767.5	**13779.0**
openthread	**7199.5**	7197.5	5935.0	5862.5	5912.0	5837.5
proj4	**6919.0**	6785.0	5365.0	5314.0	5836.5	5563.5
re2	3518.0	3533.5	3521.5	3519.0	**3544.5**	3517.0
sqlite3	35767.0	35886.5	35478.5	35845.5	35922.5	**36699.0**
vorbis	**2169.5**	2166.5	2167.5	2168.0	2168.0	2168.0
woff2	1858.0	1875.5	1934.0	1934.0	**1936.5**	1871.5

JIGSAW over Z3. Both configurations use AFL++ (commit `70bf4b4` with the default build and fuzz options[5]) for hybrid fuzzing. The experiment is conducted by Google on its cloud.

Out of 13 fuzzers (11 state-of-the-art and 2 from us), JIGSAW is 1st by average score and 1st (tied) by average rank, Z3 is 2rd by average score, and 1st by average rank. For median coverage, JIGSAW leads in 4 programs, Z3 leads in 3 programs, and AFL++ leads in 2 programs.

We also compared JIGSAW's performance with other concolic executors based on their publicly available experiment report[6]. Table 4.9 shows the result. We can see that JIGSAW can outperform other CE engines including SYMCC [53], SymQEMU [54], and Fuzzolic [9].

[5] `https://github.com/google/fuzzbench/blob/master/fuzzers/aflplusplus/fuzzer.py`
[6] `https://www.fuzzbench.com/reports/experimental/2021-07-03-symbolic/index.html`

Analysis. Because our hybrid fuzzer with JIGSAW did not outperform all other tools, including AFL++ across all benchmarks in end-to-end fuzzing, we analyzed the results to figure out the reason. The most important factor is the ability to track branches that can be affected by the inputs. Specifically, our constraint collector performs instrumentation during the compile time so it cannot collect and update path constraints in uninstrumented third-party libraries and across system calls (*e.g.*, when the input is written to another file and read back). As a result, it may try to flip fewer branches than runtime-instrumentation-based tools like QSYM [77] and SymQEMU [54]. In addition, all the evaluated concolic executors did not support tracking of floating-point number constraints, so they would not try to flip branches with floating-point number constraints. On the contrary, fuzzers like AFL++ can flip such branches.

The second issue is that the existing hybrid fuzzing scheme cannot fully utilize JIGSAW's fast-solving capability. Specifically, our hybrid fuzzer used the branch filter from QSYM [77] to determine whether a branch should be flipped or not. Because this filter is coarse-grained, many branches will be filtered. As a result, JIGSAW ended up idling most time of the fuzzing campaign. We believe a new hybrid fuzzing scheme is required to address this issue and leave it for future work.

Finally, the performance of a (hybrid) fuzzer is also constrained by other well-known factors, such as (1) the fuzzing harness [3, 36], which limits the upper bound of the code coverage that can be achieved (*e.g.*, all fuzzers saturated the coverage on some FuzzBench programs), and (2) scheduling (*e.g.*, which input to fuzz next and which technique (mutation or constraint solving) to apply).

Summary. Based on these three experiments, we conclude that **the answer to RQ4 is yes**: our approach can improve the performance of coverage-guided testing.

4.4.3 Threat to Validity

There are three major threats to the validity of our evaluation. First, although we tried to use a relatively large and diverse set of programs for evaluation, it cannot represent all programs, so the conclusion may not be generalizable to all programs. Similarly, because our constraint collector and JIGSAW only handle bitvector constraints, the conclusion may not be generalizable to other types of constraints SMT solvers support, such as floating-point numbers and strings. Second, the end-to-end performance of a coverage-guided testing tool depends on many aspects. Besides the speed of branch flipping, it also depends on path/seed scheduling, branch filtering, seed synchronization, randomness, etc. Although we have performed each experiment several times and used statistical tools, the result may not truly reflect the advantages and drawbacks of our approach. Finally, our prototype implementation could have bugs. During our evaluation, we have identified and fixed several bugs that led to poor coverage, but there could be more bugs that we have missed.

Chapter 5

Conclusions

The concolic executor (CE) is a powerful software testing tool but faces efficiency issues. Its performance slowdown comes from two sources: constraints collecting and constraints solving. In the thesis, I presented two systematic designs that alleviate CE's performance issues.

The first approach models the concolic execution as a form of dynamic data-flow analysis. As a result, the concolic executor can be built on top of LLVM DFSan, a highly-optimized dynamic data-flow analyzer. We built a prototype named SymSan. The evaluation results show that SymSan can significantly reduce the performance and memory consumption overhead simultaneously compared to previous works.

The second approach models the constraints solving as a search problem and pushes the search throughput to the next level. The key to achieving a high throughput is converting path constraints to pure straight-line functions. Based on this idea, we built a prototype

named JIGSAW. The evaluation results show that JIGSAW can significantly outperform existing SMT solvers.

Combined SymSan and JIGSAW, our concolic executor analyzes programs and generates test inputs significantly faster than the existing tools. It also helps to reach the same code coverage in a much shorter time compared to existing CEs and mutational fuzzers.

Bibliography

[1] D. Aitel. An introduction to spike, the fuzzer creation kit. *presentation slides*, 1, 2002.

[2] C. Aschermann, S. Schumilo, T. Blazytko, R. Gawlik, and T. Holz. Redqueen: Fuzzing with input-to-state correspondence. In *Annual Network and Distributed System Security Symposium (NDSS)*, 2019.

[3] D. Babić, S. Bucur, Y. Chen, F. Ivančić, T. King, M. Kusano, C. Lemieux, L. Szekeres, and W. Wang. Fudge: fuzz driver generation at scale. In *ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, 2019.

[4] M. Böhme, V. Manes, and S. K. Cha. Boosting fuzzer efficiency: An information theoretic perspective. In *ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, 2020.

[5] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury. Directed greybox fuzzing. In *ACM Conference on Computer and Communications Security (CCS)*, 2017.

[6] M. Böhme, V.-T. Pham, and A. Roychoudhury. Coverage-based greybox fuzzing as markov chain. In *ACM Conference on Computer and Communications Security (CCS)*, 2016.

[7] M. Böhme, L. Szekeres, and J. Metzman. On the reliability of coverage-based fuzzer benchmarking. In *International Conference on Software Engineering (ICSE)*, 2022.

[8] L. Borzacchiello, E. Coppa, and C. Demetrescu. Fuzzing symbolic expressions. In *International Conference on Software Engineering (ICSE)*, 2021.

[9] L. Borzacchiello, E. Coppa, and C. Demetrescu. Fuzzolic: mixing fuzzing and concolic execution. *Computers & Security*, page 102368, 2021.

[10] E. Bounimova, P. Godefroid, and D. Molnar. Billions and billions of constraints: Whitebox fuzz testing in production. In *International Conference on Software Engineering (ICSE)*, 2013.

[11] C. Cadar, D. Dunbar, and D. 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.

[12] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. Exe: automatically generating inputs of death. In *ACM Conference on Computer and Communications Security (CCS)*, 2006.

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

[14] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley. Unleashing mayhem on binary code. In *IEEE Symposium on Security and Privacy (Oakland)*, 2012.

[15] S. K. Cha, M. Woo, and D. Brumley. Program-adaptive mutational fuzzing. In *IEEE Symposium on Security and Privacy (Oakland)*, 2015.

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

[17] P. Chen, J. Liu, and H. Chen. Matryoshka: Fuzzing deeply nested branches. In *ACM Conference on Computer and Communications Security (CCS)*, 2019.

[18] Y. Chen, P. Li, J. Xu, S. Guo, R. Zhou, Y. Zhang, T. Wei, and L. Lu. Savior: Towards bug-driven hybrid testing. In *IEEE Symposium on Security and Privacy (Oakland)*, 2020.

[19] V. Chipounov, V. Kuznetsov, and G. 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.

[20] J. Choi, J. Jang, C. Han, and S. K. Cha. Grey-box concolic testing on binary code. In *International Conference on Software Engineering (ICSE)*, 2019.

[21] L. De Moura and N. Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2008.

[22] B. Dutertre. Yices 2.2. In *International Conference on Computer Aided Verification (CAV)*. Springer, 2014.

[23] M. Eddington. Peach fuzzer platform. `http://www.peachfuzzer.com/products/peach-platform/`, 2011.

[24] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse. Afl++: Combining incremental steps of fuzzing research. In *USENIX Workshop on Offensive Technologies (WOOT)*, 2020.

[25] S. Gan, C. Zhang, P. Chen, B. Zhao, X. Qin, D. Wu, and Z. Chen. Greyone: Data flow sensitive fuzzing. In *USENIX Security Symposium (Security)*, 2019.

[26] S. Gan, C. Zhang, X. Qin, X. Tu, K. Li, Z. Pei, and Z. Chen. Collafl: Path sensitive fuzzing. In *IEEE Symposium on Security and Privacy (Oakland)*, 2018.

[27] V. Ganesh and D. L. Dill. A decision procedure for bit-vectors and arrays. In *International Conference on Computer Aided Verification (CAV)*, 2007.

[28] P. Godefroid, A. Kiezun, and M. Y. Levin. Grammar-based whitebox fuzzing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2008.

[29] P. Godefroid, N. Klarlund, and K. Sen. Dart: directed automated random testing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2005.

[30] P. Godefroid, M. Y. Levin, and D. A. Molnar. Automated whitebox fuzz testing. In *Annual Network and Distributed System Security Symposium (NDSS)*, 2008.

[31] P. Godefroid, H. Peleg, and R. Singh. Learn&fuzz: Machine learning for input fuzzing. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2017.

[32] Google. TCMalloc. `https://github.com/google/tcmalloc`.

[33] Google. honggfuzz. `https://github.com/google/honggfuzz`, 2010.

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

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

[36] K. Ispoglou, D. Austin, V. Mohan, and M. Payer. FuzzGen: Automatic fuzzer generation. In *USENIX Security Symposium (Security)*, 2020.

[37] lafintel. Circumventing fuzzing roadblocks with compiler transformations. `https://lafintel.wordpress.com/`, 2016.

[38] C. Lemieux, R. Padhye, K. Sen, and D. Song. Perffuzz: automatically generating pathological inputs. In *International Symposium on Software Testing and Analysis (ISSTA)*, 2018.

[39] C. Lemieux and K. Sen. Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2018.

[40] D. Liew, C. Cadar, A. F. Donaldson, and J. R. Stinnett. Just fuzz it: solving floating-point constraints using coverage-guided fuzzing. In *ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, 2019.

[41] LLVM. Sanitizer coverage. `https://clang.llvm.org/docs/SanitizerCoverage.html`, 2017.

[42] V. J. Manès, S. Kim, and S. K. Cha. Ankou: Guiding grey-box fuzzing towards combinatorial difference. In *International Conference on Software Engineering (ICSE)*, 2020.

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

[44] B. P. Miller, L. Fredriksen, and B. So. An empirical study of the reliability of unix utilities. *Communications of the ACM*, 33(12):32–44, 1990.

[45] A. Niemetz and M. Preiner. Bitwuzla at the SMT-COMP 2020. *CoRR*, abs/2006.01621, 2020.

[46] A. Niemetz and M. Preiner. Bitwuzla at the smt-comp 2021. `https://smt-comp.github.io/2021/system-descriptions/Bitwuzla.pdf`, 2021.

[47] R. Padhye, C. Lemieux, K. Sen, M. Papadakis, and Y. Le Traon. Semantic fuzzing with zest. In *International Symposium on Software Testing and Analysis (ISSTA)*, 2019.

[48] R. Padhye, C. Lemieux, K. Sen, L. Simon, and H. Vijayakumar. Fuzzfactory: domain-specific fuzzing with waypoints. In *Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2019.

[49] A. Pandey, P. R. G. Kotcharlakota, and S. Roy. Deferred concretization in symbolic execution via fuzzing. In *International Symposium on Software Testing and Analysis (ISSTA)*, 2019.

[50] H. Peng, Y. Shoshitaishvili, and M. Payer. T-fuzz: fuzzing by program transformation. In *IEEE Symposium on Security and Privacy (Oakland)*, 2018.

[51] T. Petsios, J. Zhao, A. D. Keromytis, and S. Jana. Slowfuzz: Automated domain-independent detection of algorithmic complexity vulnerabilities. In *ACM Conference on Computer and Communications Security (CCS)*, 2017.

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

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

[54] S. Poeplau and A. Francillon. SymQEMU: Compilation-based symbolic execution for binaries. In *Annual Network and Distributed System Security Symposium (NDSS)*, 2021.

[55] L. Project. LLVM language reference manual. `https://llvm.org/docs/LangRef.html`.

[56] M. Rajpal, W. Blum, and R. Singh. Not all bytes are equal: Neural byte sieve for fuzzing. *arXiv preprint arXiv:1711.04596*, 2017.

[57] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos. Vuzzer: Application-aware evolutionary fuzzing. In *Annual Network and Distributed System Security Symposium (NDSS)*, 2017.

[58] J. Ruderman. Introducing jsfunfuzz. `http://www.squarefree.com/2007/08/02/introducing-jsfunfuzz/`, 2007.

[59] S. Schumilo, C. Aschermann, A. Abbasi, S. Wörner, and T. Holz. Nyx: Greybox hypervisor fuzzing using fast snapshots and affine types. In *30th {USENIX} Security Symposium ({USENIX} Security 21)*, 2021.

[60] S. Schumilo, C. Aschermann, R. Gawlik, S. Schinzel, and T. Holz. kAFL: Hardware-assisted feedback fuzzing for os kernels. In *USENIX Security Symposium (Security)*, 2017.

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

[62] K. Serebryany. Continuous fuzzing with libfuzzer and addresssanitizer. In *IEEE Cybersecurity Development (SecDev)*. IEEE, 2016.

[63] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov. Addresssanitizer: A fast address sanity checker. In *USENIX Annual Technical Conference (ATC)*, 2012.

[64] D. She, K. Pei, D. Epstein, J. Yang, B. Ray, and S. Jana. Neuzz: Efficient fuzzing with neural program learning. In *IEEE Symposium on Security and Privacy (Oakland)*, 2019.

[65] S. Shen, S. Shinde, S. Ramesh, A. Roychoudhury, and P. Saxena. Neuro-symbolic execution: Augmenting symbolic execution with neural constraints. In *Annual Network and Distributed System Security Symposium (NDSS)*, 2019.

[66] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, and C. Kruegel. Sok:(state of) the art of war: Offensive techniques in binary analysis. In *IEEE Symposium on Security and Privacy (Oakland)*, 2016.

[67] L. Szekeres. *Memory corruption mitigation via software hardening and bug-finding*. PhD thesis, Stony Brook University, 2017.

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

[69] D. Vyukov. Syzkaller: an unsupervised, coverage-guided kernel fuzzer, 2019.

[70] D. Wang, Z. Zhang, H. Zhang, Z. Qian, S. V. Krishnamurthy, and N. Abu-Ghazaleh. SyzVegas: Beating kernel fuzzing odds with reinforcement learning. In *USENIX Security Symposium (Security)*, 2021.

[71] J. Wang, B. Chen, L. Wei, and Y. Liu. Skyfire: Data-driven seed generation for fuzzing. In *IEEE Symposium on Security and Privacy (Oakland)*, 2017.

[72] J. Wang, C. Song, and H. Yin. Reinforcement learning-based hierarchical seed scheduling for greybox fuzzing. In *Annual Network and Distributed System Security Symposium (NDSS)*, 2021.

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

[74] W. Xu, S. Kashyap, C. Min, and T. Kim. Designing new operating primitives to improve fuzzing performance. In *ACM Conference on Computer and Communications Security (CCS)*, 2017.

[75] W. You, X. Wang, S. Ma, J. Huang, X. Zhang, X. Wang, and B. Liang. Profuzzer: On-the-fly input type probing for better zero-day vulnerability discovery. In *IEEE Symposium on Security and Privacy (Oakland)*, 2019.

[76] T. Yue, P. Wang, Y. Tang, E. Wang, B. Yu, K. Lu, and X. Zhou. Ecofuzz: Adaptive energy-saving greybox fuzzing as a variant of the adversarial multi-armed bandit. In *USENIX Security Symposium (Security)*, 2020.

[77] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim. Qsym: A practical concolic execution engine tailored for hybrid fuzzing. In *USENIX Security Symposium (Security)*, 2018.

[78] M. Zalewski. American fuzzy lop.(2014). `http://lcamtuf.coredump.cx/afl`, 2014.

[79] Y. Zheng, A. Davanian, H. Yin, C. Song, H. Zhu, and L. Sun. Firm-afl: high-throughput greybox fuzzing of iot firmware via augmented process emulation. In *USENIX Security Symposium (Security)*, 2019.

Appendix

Concolic Execution without Solving. Table 5.1 shows the execution time of SymSan, SymCC, and SymQEMU to collect symbolic constraints without solving (section 3.4.2).

Effectiveness of the Two Additional Optimizations. Figure 5.1 shows the execution time distribution of (1) native execution, (2) SymSan *without* expression deduplication *and* load/store optimization, (3) SymSan without load/store optimization, and (4) full-fledge SymSan. As we can see, both optimization techniques can help reduce the execution time.

Statistics of Collected Constraints. Figure 5.2 shows (1) the maximum number of tracked expressions (in number of AST nodes).

Table 5.1: 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**

We checked the total number of branch predicates that fell back to optimistic solving (because of timeout), as an indication of the difficulties of the constraints. For SymCC, the number is 2166. For SymSan, the number is 2461.

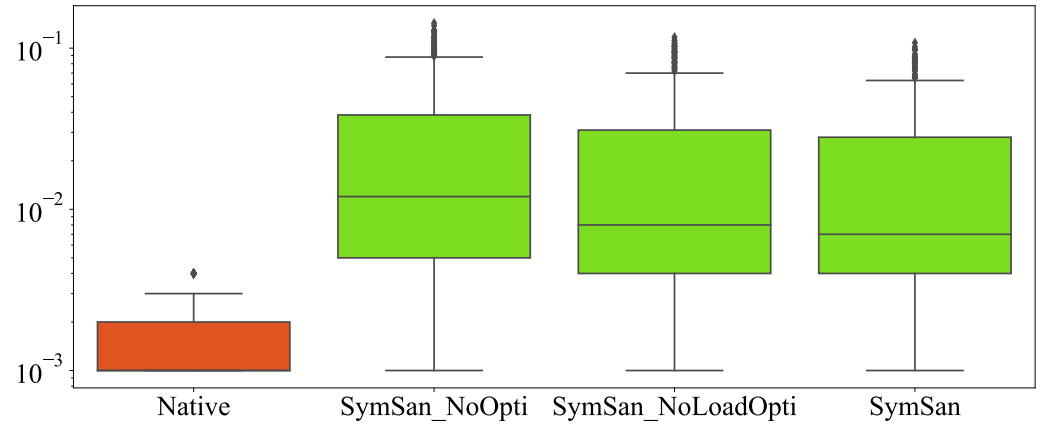

Figure 5.1: Execution time for the real-world programs. The figure is drawn in logarithmic scale. SymSan-NoOpti is SymSan without expressions deduplication and load/store optimization. SymSan-NoLoad is without load/store optimization).

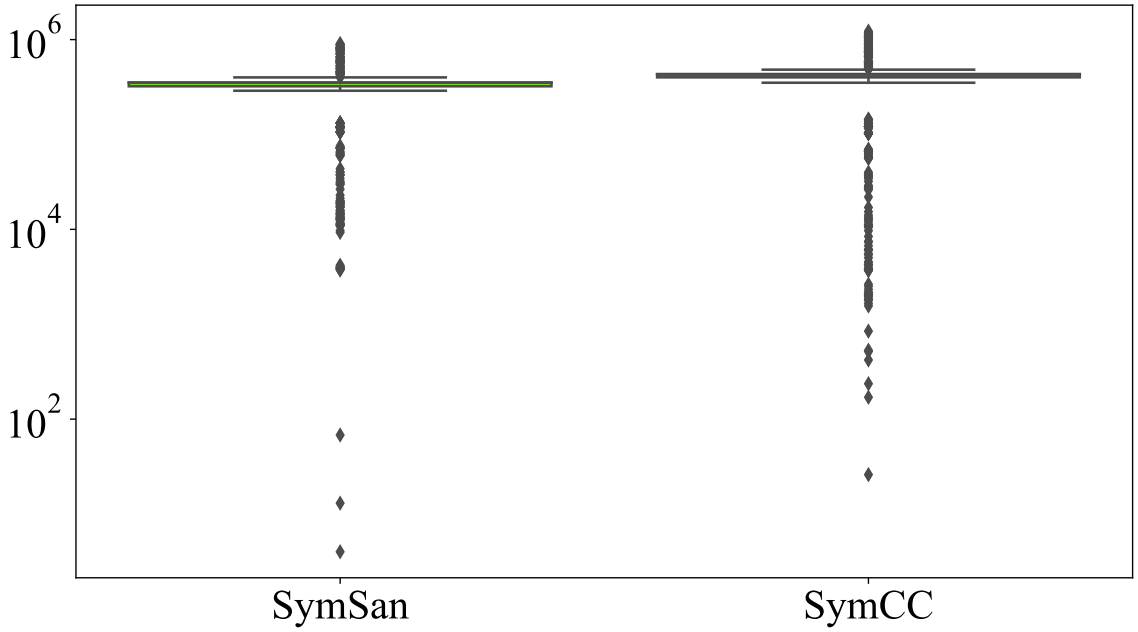

Figure 5.2: Maximum number of AST nodes tracked by SymSan and SymCC