JIGSAW: Efficient and Scalable Path Constraints Fuzzing

Ju Chen, Jinghan Wang, Chengyu Song, Heng Yin
Computer Science and Engineering Department
University of California, Riverside

***Abstract*—Coverage-guided testing has shown to be an effective way to find bugs. If we model coverage-guided testing as a search problem (i.e., finding inputs that can cover more branches), then its efficiency mainly depends on two factors: (1) the accuracy of the searching algorithm and (2) the number of inputs that can be evaluated per unit time. Therefore, improving the search throughput has shown to be an effective way to improve the performance of coverage-guided testing.**

In this work, we present a novel design to improve the search throughput: by evaluating newly generated inputs with JIT-compiled path constraints. This approach allows us to significantly improve the single thread throughput as well as scaling to multiple cores. We also developed several optimization techniques to eliminate major bottlenecks during this process. Evaluation of our prototype JIGSAW shows that our approach can achieve three orders of magnitude higher search throughput than existing fuzzers and can scale to multiple cores. We also find that with such high throughput, a simple gradient-guided search heuristic can solve path constraints collected from a large set of real-world programs faster than SMT solvers with much more sophisticated search heuristics. Evaluation of end-to-end coverage-guided testing also shows that our JIGSAW-powered hybrid fuzzer can outperform state-of-the-art testing tools.

I. INTRODUCTION

Exploring a program's execution space is essential for finding bugs and security flaws in software. Starting from a set of initial inputs (*a.k.a.* seeds) to the program, an automatic testcase generation technique is expected to quickly find more inputs that can exercise new states that are not covered before, such that more and more behaviors can be revealed. Because bugs tend to reside in less-tested code, coverage-guided testing, which aims to cover as much code as possible, is a popular and effective way to find bugs.

Fuzzing and symbolic execution are two representative testcase generation techniques. Fuzzing [35, 65, 78] discovers new coverage via random search: a new input is randomly generated or mutated from an old input, which is then tested with the program under test (PUT) to check whether one or more new branches are covered. Symbolic execution [10–15, 19, 31, 32, 56, 57, 64, 77], on the other hand, performs more systematic exploration: it collects path constraints during the execution and uses a constraint solver like a satisfiability modulo theories (SMT) solver [4, 21, 22, 29, 49, 50] to generate new input that can flip branches along execution paths.

If we model the coverage-guided testing as a search problem that tries to *find an input that can satisfy a target branch's predicate*, then the efficiency of a coverage-guided testing tool can be measured using the *branch flip rate* (*i.e.*, how many branches are flipped per unit time), which is determined by two factors: search throughput and search accuracy. Search throughput represents how many inputs can be tried per unit time, and search accuracy specifies how likely a newly generated input can flip a branch. Obviously, the branch flip rate is a product of search throughput and search accuracy. Overall, a tool with a higher branch flipping rate can achieve the same coverage faster and is more likely to find more bugs [5].

TABLE I: Comparison of branch flipping strategies.

Tools	Searching target	Form of target	Core search alg.
AFL [78]	Whole program	Native code	Random search
Angora [17]	Whole program	Native code	Gradient-guided search
SymCC [56]	Path constraint	SMT formula	SMT solving (DPLL)
Fuzzolic [9]	Path constraint	SMT formula	Fuzzing heuristics
JIGSAW	Path constraint	**JIT'ed code**	Gradient-guided search

In this work, we investigate a new point in the design space of coverage-guided testing (Table I). Our design goal is to improve the search throughput. Our key insight is that evaluating a new test input with path constraints in the form of native functions, produced by Just-In-Time (JIT) compilation, can be much faster than both traditional approaches. Specifically, when comparing with fuzzers that evaluate a new input with the entire PUT, we have several observations. (1) Invoking a set of native functions is orders of magnitude faster than executing the whole program. (2) Since a branch's path constraints do not update any global state, the JIT'ed functions are side-effect free (*i.e.*, pure). So, evaluating new inputs with them avoids expensive state reset processes like forking a new process. (3) When evaluating a new test input with functions, the input can be passed through registers and memory instead of the file system, which eliminates the file system bottleneck [74]. (4) Because every JIT'ed function is pure (*i.e.*, independent of each other), we can linearly scale the fuzzing threads to multiple cores or machines [36] without worrying about data races and synchronization. (5) The JIT'ed path constraints are usually free of branches, which makes it easier for modern processors to exploit instruction-level parallelism (*i.e.*, no mis-speculation) and to adopt SIMD (Single Instruction Multiple Data) instructions to further improve the throughput via data parallelization [25]. As a result, our approach can improve both the sequential throughput (properties 1-3) and parallel

throughput (properties 3-5).

Compared to other solvers, which also evaluate a new input with path constraints, including JIT-compiled constraints [40, 45, 53], our approach also features several optimizations to improve the search throughput. First, to minimize the cost of JIT compilation, we used an in-memory JIT engine and implemented a constraint to JIT'ed function cache. More importantly, we perform constraint normalization before accessing the cache. We observe that many path constraints essentially perform the same check over different inputs. In other words, the abstract syntax trees (AST) of these path constraints only differ at the leaf nodes, so they can be solved using the same JIT'ed function. For instance, we can reuse the same JIT'ed function

```
gt(x, y) { return y - x; }
```

to solve constraints like $a > 10$, $b > 20$, $30 > c$, etc. This normalization process significantly increases the cache hit rate. Second, we reduced the number of invoked native functions by only evaluating those that will be affected by the new input. Finally, we reduced the use of locks to allow a more scalable parallel search.

To validate this idea, we implement a prototype, called JIGSAW (**J**ust-**i**n-Time **G**radient descent **S**earch for **A**ns**W**ers). It compiles path constraints collected from the target program into a set of native functions using the JIT engine from LLVM. Then it performs a gradient-guided search [17, 18], a relatively simple local search heuristic, to find an input that can flip the corresponding branch. Our evaluation of a set of popular applications shows that JIGSAW can achieve an average search throughput of 637.2K inputs/sec when fuzzing path constraints with data dependencies (*a.k.a.* nested branch constraints) using a single thread, and can scale to 12.5M inputs/sec using 48 threads. The corresponding branch flipping rate is about 588.9 branches/sec using a single thread and can scale to 11.3K branches/sec using 48 threads. When solving last-branch constraints without dependencies (*a.k.a.* optimistic solving), the search throughput scales from 3.1M inputs/sec with a single thread to 74.7M with 48 threads. The corresponding branch flipping rate is about 35.7K and 860.0K branches/sec, respectively. For comparison, on `libpng`, a recent work [74] on improving fuzzing throughput reported a throughput around 6.5M inputs/sec with libFuzzer using 120 cores; JIGSAW can achieve 18.1M inputs/sec with 48 cores. Interestingly, such high search throughput allowed JIGSAW to solve path constraints collected from real-world applications faster than SMT solvers powered with much more sophisticated search strategies.

To evaluate JIGSAW's impact on end-to-end coverage-guided testing, we also implemented a hybrid fuzzer with JIGSAW as the path constraint solver. The evaluation results showed that the high branch flipping rate of JIGSAW allowed our hybrid fuzzer to achieve the same code coverage faster than existing fuzzers and symbolic executors.

In summary, we make the following contributions:

- We designed a novel approach that improves the branch flipping rate of automated test generation.
- We implemented a prototype JIGSAW and open-sourced it (https://github.com/R-Fuzz/jigsaw).
- We evaluated our prototype with a set of real-world applications. The results showed that our approach can significantly improve the search throughput, which enables better performance in coverage-guided testing.
- We released the path constraints we collected.

II. BACKGROUND

In this section, we first provide an overview of automated test generation techniques, including modern feedback-guided fuzzers and symbolic execution. Then we discuss the important factors that affect their efficiency.

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 [47], 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 [1, 23, 30, 51, 61] or learning based [33, 59, 72]. Mutational fuzzers generally adopt two genetic operations: random mutation and crossover [35, 47, 65, 78]. The first generation of fuzzers was blackbox fuzzers [1, 23, 47], 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 [2, 6–8, 17, 18, 20, 27, 35, 42, 46, 55, 65, 69, 75, 76, 78]. 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 [52].

White-box fuzzers and symbolic/concolic executors [10–15, 19, 31, 32, 56, 57, 64, 77] 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) [14]. 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.

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 [16, 76] and numerous scheduling algorithms have been proposed to improve the efficiency of fuzzers [6–8, 28, 43, 60, 70, 71, 76].

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 [18]. The efficiency of this step depends on two factors. The first factor is the search algorithm. Off-the-shelf fuzzers [35, 65, 78] 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 [2, 54, 60, 73]. A more general solution is to measure progress and perform a directed search, such as splitting branches [41], using gradient-guided search [17, 18, 66, 67], binary search [20], genetic algorithms [27], or simulated annealing [68]. 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 [9, 45, 53]. 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 [63] 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 [62] 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 [10, 56, 57, 77].

III. Our Approach

A. 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 are 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 mean we can linearly scale 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 arguments via registers or memory thus avoiding 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, 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.

B. Overview

Figure 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.

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 5. In step ❶, we use a symbolic

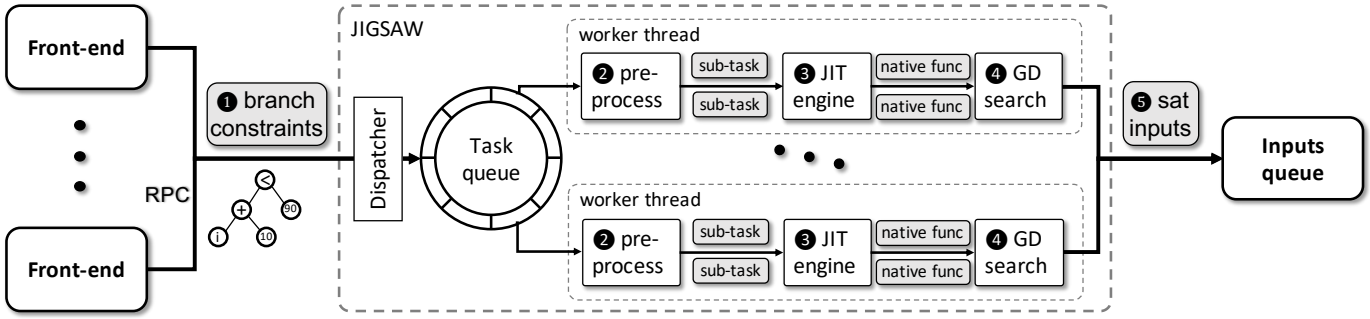


Fig. 1: Overview of JIGSAW

```
1 bool test(i) { return i < 13; }
2 void main() {
3   unsigned int x;
4   read(0, &x, sizeof(x));
5   if (x > 8 && test(x))
6     assert(0);
7 }
```

Listing 1: A running example to demonstrate the workflow.

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 the following.

```
1  land(
2    ugt(read(0, 32), constant(8, 32)),
3    ult(read(0, 32), constant(13, 32))
4  )
```

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 values: `true` and `false`. The result is as follows.

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

After preprocessing, in step ❸, we compile each sub-task into a function in LLVM IR. Note that unlike previous work [40, 45, 53], these functions are generated in memory using LLVM's C++ API instead of writing to files. We then use LLVM's JIT engine to compile each IR function to a native function. In this step, 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 may reduce the overall branch flipping rate because compilation is much more expensive than fuzzing (see §VI for more details).

In step ❹, we plug two JIT'ed functions into the gradient-guided search algorithm from Matryoshka [18] to search for a satisfying `x`. This algorithm can jointly solve conjunctions of sub-tasks.

C. Challenges

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

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 §VI). 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.

D. 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

```
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 2: AST node for function cache lookup.

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 [17]), 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 [48], 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.

IV. JIGSAW

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

A. 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 (§V). Similar to SymCC [56], our engine collects path constraints at the LLVM IR [58] level. The collected path constraints are then passed to JIGSAW through shared memory.

AST for Cache Lookup. Listing 2 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 [18]. 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.

B. 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 (§IV-D).

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 $\geq$).

Arguments Mapping. To maximize the reuse of JIT'ed functions and minimize the compilation time (§IV-E), 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.

TABLE II: Transforming a comparison operation into a distance-based loss function. a and b are arbitrary ASTs/expressions, ϵ is a small positive value, e.g., integer 1.

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

C. 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 [17, 18, 67]. Table II 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.

D. Solving

To search for a satisfying input, we use the gradient-guided search algorithm from Matryoshka [18], 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.

E. 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 §III, 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 [34] from Google to reduce contentions caused by `malloc` and `free`.

V. 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 [17]. 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 [17].

VI. 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.

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

TABLE III: 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 IV: Solving capability comparison. The timeout of JIGSAW is set to one million iterations (JIGSAW-1M). The timeout of Z3 and Bitwuzla is set to 60 seconds (Z3-60s, baseline). The timeout of Bitwuzla local search (LS) only mode is also set to 1M updates. Nested means the percentage of solved nested branch constraints. Single means the percentage of solved last branch constraints.

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

Dataset. We used two datasets in our evaluation. The first dataset includes 14 real-world programs (Table III). We use this dataset to answer RQ1, RQ2, and RQ3. To answer RQ4, our main dataset is the Google Fuzzbench [37]. To compare with fuzzers that are not supported by Fuzzbench, we use part of our first dataset.

A. Constraint Solving Performance

To evaluate the solving performance, we collected about 10 million path constraints from 14 real-world programs (Table III). We first use AFL to fuzz the target programs for 48 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 §VI-B). 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 C/C++ bindings, including Z3 [21], STP [29], Yices2 [22], and Bitwuzla [48]. For Bitwuzla, we evaluated two different modes: (1) the configuration that won the SMT-COMP 2021 [49]

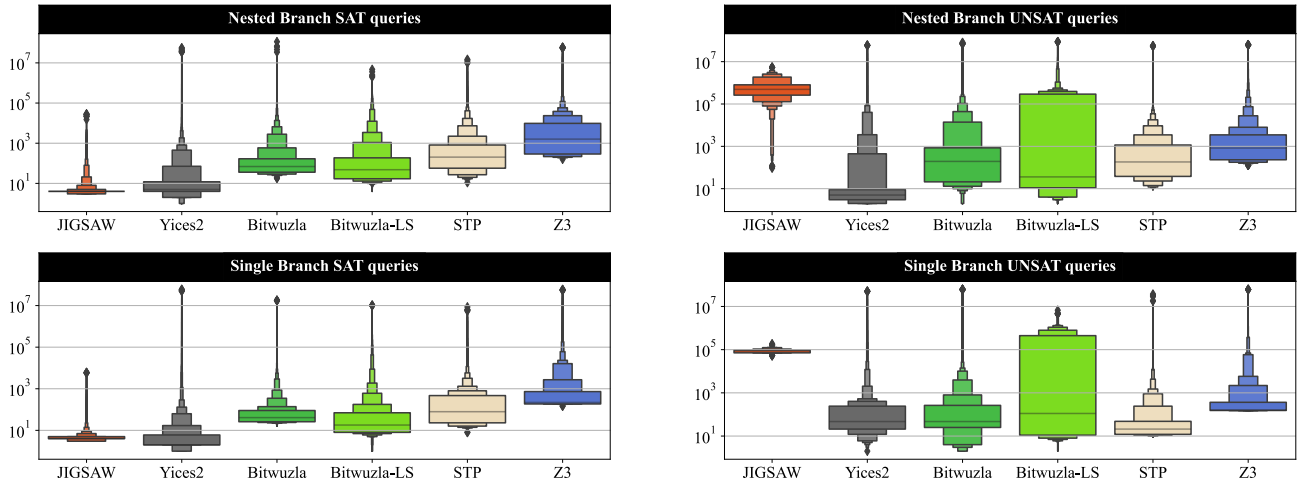


Fig. 2: Constraints processing time distribution (in micro-seconds). Because of the large data set, we use boxenplot [38] for illustration. The centerline indicates the median. The first two boxes surrounding the center line contain 50% of the data. Each successive level outward contains half of the remaining data. We use award-winning configurations [49] for Bitwuzla. Bitwuzla-LS is Bitwuzla in local search only mode.

(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 IV. 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 (§IV-D) 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 II). 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 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 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 V: The throughput (number of tried inputs per second) of JIGSAW (JIGSAW-1K) and Bitwuzla (BZLA-LS-100K) in a single-threaded execution. The first half of the table shows the results of nested branch constraints, and the second half shows the results of single branch constraints.

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

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 V 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 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 VI, 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 VII 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 (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 [53, 61]). 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 VIII. 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

TABLE VI: The branch flipping rate of single thread JIGSAW and comparison with popular SMT solvers. BZLA is the abbreviation of Bitwuzla. The first half of the table shows the results of nested branch constraints; the second half shows the results of single branch constraints.

Program	Nested Branch Constraints						Single Branch Constraints					
	Yices2	BZLA-LS-100K	BZLA-6MS	STP	Z3-50MS	JIGSAW-1K	Yices2	BZLA-LS-100K	BZLA-6MS	STP	Z3-50MS	JIGSAW-1K
objdump	134.2	19.3	23.9	38.6	22.5	300.0	21.9 K	0.6 K	0.9 K	2.4 K	0.5 K	73.4 K
size	436.1	54.3	114.1	132.7	54.4	1296.0	13.9 K	0.6 K	0.7 K	1.1 K	0.6 K	34.7 K
nm	4540.4	281.4	486.4	754.7	294.5	6679.5	39.7 K	1.3 K	1.4 K	4.4 K	0.5 K	41.3 K
readelf	18.1	22.7	52.6	82.5	42.9	433.9	11.5 K	0.4 K	0.6 K	1.1 K	1.0 K	38.7 K
libpng	870.7	26.7	53.0	269.6	120.3	736.1	14.4 K	0.9 K	0.9 K	1.2 K	0.1 K	28.8 K
tiff2pdf	220.9	54.8	55.2	44.1	24.6	155.5	40.5 K	1.5 K	1.9 K	10.2 K	1.2 K	98.8 K
file	104.1	26.4	26.8	43.5	14.0	267.7	22.4 K	0.9 K	1.1 K	2.9 K	0.4 K	61.0 K
tcpdump	1298.6	155.1	147.1	435.5	92.5	1850.1	23.7 K	0.7 K	1.1 K	4.0 K	0.5 K	27.7 K
openssl	2.4	29.9	44.3	46.6	21.7	303.4	6.6	1.1 K	1.1 K	0.6 K	0.8 K	16.3 K
sqlite3	25429.2	1448.3	1338.0	2622.4	896.5	12510.5	70.2 K	2.3 K	2.3 K	7.7 K	2.0 K	156.2 K
vorbis	93.1	14.6	23.4	36.1	7.0	168.3	1.2 K	0.3 K	0.4 K	2.0 K	0.2 K	8.0 K
mbedtls	42.2	6.3	6.8	12.7	5.3	98.4	4.5 K	0.8 K	0.4 K	0.3 K	16.3	14.4 K
libxml2	3531.4	46.4	488.5	562.0	191.6	3925.3	44.0 K	2.3 K	2.1 K	5.7 K	0.3 K	113.7 K
libjpeg-turbo	2.1	4.1	2.6	4.5	4.7	38.3	92.4	192.6	226.5	214.7	53.7	8.2 K
Geomean	204.4	40.3	61.0	100.1	41.0	588.9	6.8 K	0.8 K	0.9 K	1.2 K	0.3 K	35.7 K

TABLE VII: Accumulated solving time breakdown of JIGSAW, when solving all constraints using 1000 iterations as the timeout.

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

TABLE VIII: 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

comparison, we also tried multi-threaded Z3 where each thread uses a separate Z3 context and solver.

Figure 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

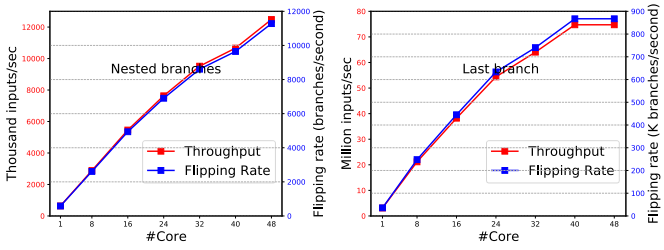


Fig. 3: Average search throughput and branch flipping rate of JIGSAW on multiple cores.

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.

B. 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 [56], SymQEMU [57], and Fuzzolic [10]. 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 [57], 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

TABLE IX: Comparison of concolic execution engines on flipping all symbolic branches along a single execution trace. The top half shows the execution time, the bottom half shows the basic-block coverage measured by SanitizerCoverage.

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

setting used by other concolic executors [56, 57, 77] and 50ms timeout is the setting that offers a similar solving capability as JIGSAW-1K. The next one is Angora [17]. 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 [17] 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 [56], 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 [10], with Fuzzy-Sat [9], 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 IX 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 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.

Local Fuzzing. Next, we evaluated three fuzzers that are not supported by Fuzzbench. The first one is Angora [17]. 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 [66], 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 [10] with Fuzzy-Sat [9] as the solver. Finally, we also included AFL++ [26] (3.12c with `cmplog` enabled), the state-of-the-art fuzzer.

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

TABLE X: Comparing JIGSAW with other state-of-the-art symbolic executors based on their publicly available Fuzzbench results. The metric is median coverage reached in 24 hours. We show the results of 16 programs where all tools generate valid results. JIGSAW takes the lead in 7 out of 16 programs among 5 hybrid fuzzers (two from us) and 1 non-hybrid fuzzer AFL++.

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

For comparison, we used the same strategy as Neuzz [66]. 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 III: `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 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.

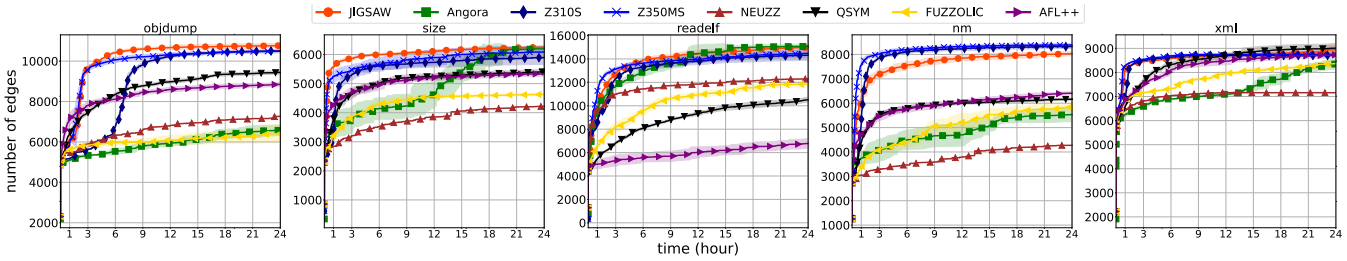


Fig. 4: Edge coverage growth over time for local fuzzing.

Fuzzbench. Next, we compared JIGSAW with other popular fuzzers on Google the Fuzzbench dataset [37]. 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 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. Due to the page limit, we only provide a summary in this section. The more detailed results are presented in §X.

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 X shows the result. We can see that JIGSAW can outperform other CE engines including SYMCC [56], SymQEMU [57], and Fuzzolic [10].

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 [57]. 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, 39], 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.

C. 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.

VII. DISCUSSION

In this section, we discuss the limitations of our current prototype and potential future works.

Faster JIT Engine. In our current prototype, LLVM's JIT engine is a main performance bottleneck (*i.e.*, JIGSAW spends similar time on JIT as time on solving). We expect a faster JIT engine that can directly compile the AST into native code (*e.g.*, the tiny code generator from QEMU) can help further improve the performance and scalability of JIGSAW.

[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

Better Task Scheduling. Different constraints require different numbers of iterations to solve. Right now we use a single timeout for all tasks. As a result, if we make this timeout too large, more complex constraints may block simpler constraints and reduce the total branch flipping rate; on the other hand, if we make this timeout too small, we may not be able to solve those more complex constraints. We plan to adopt an OS-style scheduler to balance this. Essentially, we can prioritize simpler constraints to be solved first but still allow more complex constraints to run for a much longer time.

String and Floating-point Operations. Due to the limitation of concolic execution engines (both QSYM [77] and our DFSan-based executor), our current prototype neither collects nor solves constraints involving string and floating-point operations. However, as shown in previous work [45, 53], even random-mutation-based fuzzing can be more efficient than SMT solvers in solving constraints involving string and floating-point operations. We believe it is possible to apply their algorithms to JIGSAW. We will explore this direction in future work.

Better Searching Algorithms. Since the main focus of JIGSAW is to improve the execution throughput, we did not spend too much effort on improving the search algorithm itself and simply adopted the numeric gradient descent algorithm from Angora [17]. While this algorithm is general, it is not the most efficient one [9] and does not work very well when the number of conjunct constraints is large. We plan to investigate and adopt other searching heuristics, including ones from SMT solvers to overcome this limitation.

VIII. RELATED WORK

A. Improving Search Accuracy

The main task for automated test generation is solving branch constraints. There are two main ways to improve the performance of branch constraint solving. The first way is to improve the efficiency of the solving algorithm. To solve complex constraints or constraints with tight conditions (e.g., magic bytes matching), researchers have proposed many solutions. Vuzzer [60] uses taint analysis to find magic bytes matching and solves them by copying the expected values. Steelix [44] and REDQUEEN also aim to solve magic bytes and simple input transformation but use offset inference instead of heavy dynamic taint analysis. TaintScope [73] and T-Fuzz [54] avoid complex constraints (e.g., checksum check) by patching the corresponding branches. The problem with these approaches is that they only target one or few types of constraints and thus are not very generalizable.

Angora [17] and Matryoshka [18] use taint analysis to collect input dependencies and use numerical gradient descent to solve branch constraints. Eclipser [20] uses coverage information to infer input dependencies and then uses binary search to solve branch constraints. GreyOne [27] also uses coverage information to infer input dependencies but the scope is expanded to both direct dependencies and indirect dependencies, then it uses a genetic algorithm to solve the constraints.

Whitebox fuzzers [11–15, 19, 31, 32, 64, 77] collect branch constraints as symbolic formulas and use SMT solvers to solve them. While SMT solvers are very powerful, they are not very efficient at solving constraints over floating-point and strings. To overcome this limitation, researchers have proposed using a deep neural network to simulate the target constraints and solve them using gradient descent [67]. They have also shown that random-mutation-based fuzzing can be more efficient at solving these constraints [45, 53].

JIGSAW does not aim to improve the efficiency of the solving algorithm. We simply adopted the numeric gradient descent algorithm from [17]. Our goal is to improve search throughput.

B. Improving Fuzzing Throughput

When a branch solving algorithm is fixed, another way to improve the performance of branch solving is to improve the throughput, i.e., the number of inputs that can be tried in a given period. The most straightforward way to improve fuzzing throughput is to improve parallelism. ClusterFuzz [36] uses a cluster of machines to improve fuzzing throughput. The problem, as pointed out by Xu *et al.* [74] is that running several fuzzing instances in parallel does not scale very well to multiple cores on commodity OS. To solve the bottlenecks, they proposed new OS primitives. Besides running multiple instances, another way to improve parallelism is to use SIMD instructions to do data parallelization [25]. The challenge for data parallelization, however, is conditional branches. Most solutions will end up limiting the data parallelism [24], i.e., disabling deviated data lanes. JIGSAW can use both multi-core and SIMD to improve parallelism. Compared with existing solutions, JIGSAW's approach is more scalable to multiple cores as it has fewer synchronization bottlenecks; it is also more efficient at using data parallelism because the JIT-compiled functions do not have conditional branches.

IX. CONCLUSION

In this paper, we present a novel design to improve the search throughput in automated test generation, based on a powerful insight: searching for a satisfiable input is much more efficient and scalable with path constraints than with the whole original program. Our evaluation results showed that our approach indeed can achieve a search throughput orders of magnitude higher than state-of-the-art fuzzers, which can lead to significant improvement in branch flipping rate and end-to-end coverage-guided testing.

ACKNOWLEDGMENTS

This work is supported, in part, by the National Science Foundation under Grant No. 2046026, No. 2133487, and the Office of Naval Research under Award No. N00014-17-1-2893. Any opinions, findings, conclusions, or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding agencies.

References

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

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

[3] Domagoj Babić, Stefan Bucur, Yaohui Chen, Franjo Ivančić, Tim King, Markus Kusano, Caroline Lemieux, László Szekeres, and Wei Wang. Fudge: fuzz driver generation at scale. In *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2019.

[4] Clark Barrett, Christopher L Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. Cvc4. In *International Conference on Computer Aided Verification (CAV)*, pages 171–177. Springer, 2011.

[5] Marcel Böhme and Brandon Falk. Fuzzing: On the exponential cost of vulnerability discovery. In *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2020.

[6] Marcel Böhme, Valentin Manes, and Sang Kil Cha. Boosting fuzzer efficiency: An information theoretic perspective. In *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2020.

[7] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. Directed greybox fuzzing. In *ACM Conference on Computer and Communications Security (CCS)*, 2017.

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

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

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

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

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

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

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

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

[16] Sang Kil Cha, Maverick Woo, and David Brumley. Program-adaptive mutational fuzzing. In *IEEE Symposium on Security and Privacy (Oakland)*, 2015.

[17] Peng Chen and Hao Chen. Angora: Efficient Fuzzing by Principled Search. In *IEEE Symposium on Security and Privacy (Oakland)*, 2018.

[18] Peng Chen, Jianzhong Liu, and Hao Chen. Matryoshka: Fuzzing deeply nested branches. In *ACM Conference on Computer and Communications Security (CCS)*, 2019.

[19] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. S2E: A platform for in-vivo multi-path analysis of software systems. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2011.

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

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

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

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

[24] Brandon Falk. How conditional branches work in vectorized emulation. https://gamozolabs.github.io/fuzzing/2019/10/07/vectorized_emulation_condbranch.html, 2018.

[25] Brandon Falk. Vectorized emulation: Hardware accelerated taint tracking at 2 trillion instructions per second. https://gamozolabs.github.io/fuzzing/2018/10/14/vectorized_emulation.html, 2018.

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

[27] Shuitao Gan, Chao Zhang, Peng Chen, Bodong Zhao, Xiaojun Qin, Dong Wu, and Zuoning Chen. Greyone: Data flow sensitive fuzzing. In *USENIX Security Symposium (Security)*, 2019.

[28] Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuwen Tu, Kang Li, Zhongyu Pei, and Zuoning Chen. Collafl: Path sensitive fuzzing. In *IEEE Symposium on Security and Privacy (Oakland)*, 2018.

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

[30] Patrice Godefroid, Adam Kiezun, and Michael Y Levin. Grammar-based whitebox fuzzing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2008.

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

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

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

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

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

[36] Google. Fuzzing for security. https://blog.chromium.org/2012/04/fuzzing-for-security.html, 2012.

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

[38] Heike Hofmann, Karen Kafadar, and Hadley Wickham. Letter-value plots: Boxplots for large data. Technical report, had.co.nz, 2011.

[39] Kyriakos Ispoglou, Daniel Austin, Vishwath Mohan, and Mathias Payer. FuzzGen: Automatic fuzzer generation. In *USENIX Security Symposium (Security)*, 2020.

[40] M Ammar Ben Khadra, Dominik Stoffel, and Wolfgang Kunz. gosat: floating-point satisfiability as global optimization. In *Formal Methods in Computer Aided Design (FMCAD)*, 2017.

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

[42] Caroline Lemieux, Rohan Padhye, Koushik Sen, and Dawn Song. Perffuzz: automatically generating pathological inputs. In *International Symposium on Software Testing and Analysis (ISSTA)*, 2018.

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

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

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

[46] Valentin JM Manès, Soomin Kim, and Sang Kil Cha. Ankou: Guiding grey-box fuzzing towards combinatorial difference. In *International Conference on Software Engineering (ICSE)*, 2020.

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

[48] Aina Niemetz and Mathias Preiner. Bitwuzla at the SMT-COMP 2020. *CoRR*, abs/2006.01621, 2020.

[49] Aina Niemetz and Mathias Preiner. Bitwuzla at the smt-comp 2021. https://smt-comp.github.io/2021/system-descriptions/Bitwuzla.pdf, 2021.

[50] Aina Niemetz, Mathias Preiner, and Armin Biere. Boolector 2.0. *J. Satisf. Boolean Model. Comput.*, 9(1):53–58, 2014.

[51] Rohan Padhye, Caroline Lemieux, Koushik Sen, Mike Papadakis, and Yves Le Traon. Semantic fuzzing with zest. In *International Symposium on Software Testing and Analysis (ISSTA)*, 2019.

[52] Rohan Padhye, Caroline Lemieux, Koushik Sen, Laurent Simon, and Hayawardh Vijayakumar. Fuzzfactory: domain-specific fuzzing with waypoints. In *Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2019.

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

[54] Hui Peng, Yan Shoshitaishvili, and Mathias Payer. T-fuzz: fuzzing by program transformation. In *IEEE Symposium on Security and Privacy (Oakland)*, 2018.

[55] Theofilos Petsios, Jason Zhao, Angelos D Keromytis, and Suman Jana. Slowfuzz: Automated domain-independent detection of algorithmic complexity vulnerabilities. In *ACM Conference on Computer and Communications Security (CCS)*, 2017.

[56] Sebastian Poeplau and Aurélien Francillon. Symbolic execution with symcc: Don‘t interpret, compile! In *USENIX Security Symposium (Security)*, 2020.

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

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

[59] Mohit Rajpal, William Blum, and Rishabh Singh. Not all bytes are equal: Neural byte sieve for fuzzing. *arXiv preprint arXiv:1711.04596*, 2017.

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

[61] Jesse Ruderman. Introducing jsfunfuzz. http://www.squarefree.com/2007/08/02/introducing-jsfunfuzz/, 2007.

[62] Sergej Schumilo, Cornelius Aschermann, Ali Abbasi, Simon Wörner, and Thorsten Holz. Nyx: Greybox hypervisor fuzzing using fast snapshots and affine types. In *30th {USENIX} Security Symposium ({USENIX}*

Security 21), 2021.

[63] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. kAFL: Hardware-assisted feedback fuzzing for os kernels. In *USENIX Security Symposium (Security)*, 2017.

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

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

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

[67] Shiqi Shen, Shweta Shinde, Soundarya Ramesh, Abhik Roychoudhury, and Prateek Saxena. Neuro-symbolic execution: Augmenting symbolic execution with neural constraints. In *Annual Network and Distributed System Security Symposium (NDSS)*, 2019.

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

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

[70] Daimeng Wang, Zheng Zhang, Hang Zhang, Zhiyun Qian, Srikanth V Krishnamurthy, and Nael Abu-Ghazaleh. SyzVegas: Beating kernel fuzzing odds with reinforcement learning. In *USENIX Security Symposium (Security)*, 2021.

[71] Jinghan Wang, Chengyu Song, and Heng Yin. Reinforcement learning-based hierarchical seed scheduling for greybox fuzzing. In *Annual Network and Distributed System Security Symposium (NDSS)*, 2021.

[72] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. Skyfire: Data-driven seed generation for fuzzing. In *IEEE Symposium on Security and Privacy (Oakland)*, 2017.

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

[74] Wen Xu, Sanidhya Kashyap, Changwoo Min, and Taesoo Kim. Designing new operating primitives to improve fuzzing performance. In *ACM Conference on Computer and Communications Security (CCS)*, 2017.

[75] Wei You, Xueqiang Wang, Shiqing Ma, Jianjun Huang, Xiangyu Zhang, XiaoFeng Wang, and Bin Liang. Profuzzer: On-the-fly input type probing for better zero-day vulnerability discovery. In *IEEE Symposium on Security and Privacy (Oakland)*, 2019.

[76] Tai Yue, Pengfei Wang, Yong Tang, Enze Wang, Bo Yu, Kai Lu, and Xu Zhou. Ecofuzz: Adaptive energy-saving greybox fuzzing as a variant of the adversarial multi-armed bandit. In *USENIX Security Symposium (Security)*, 2020.

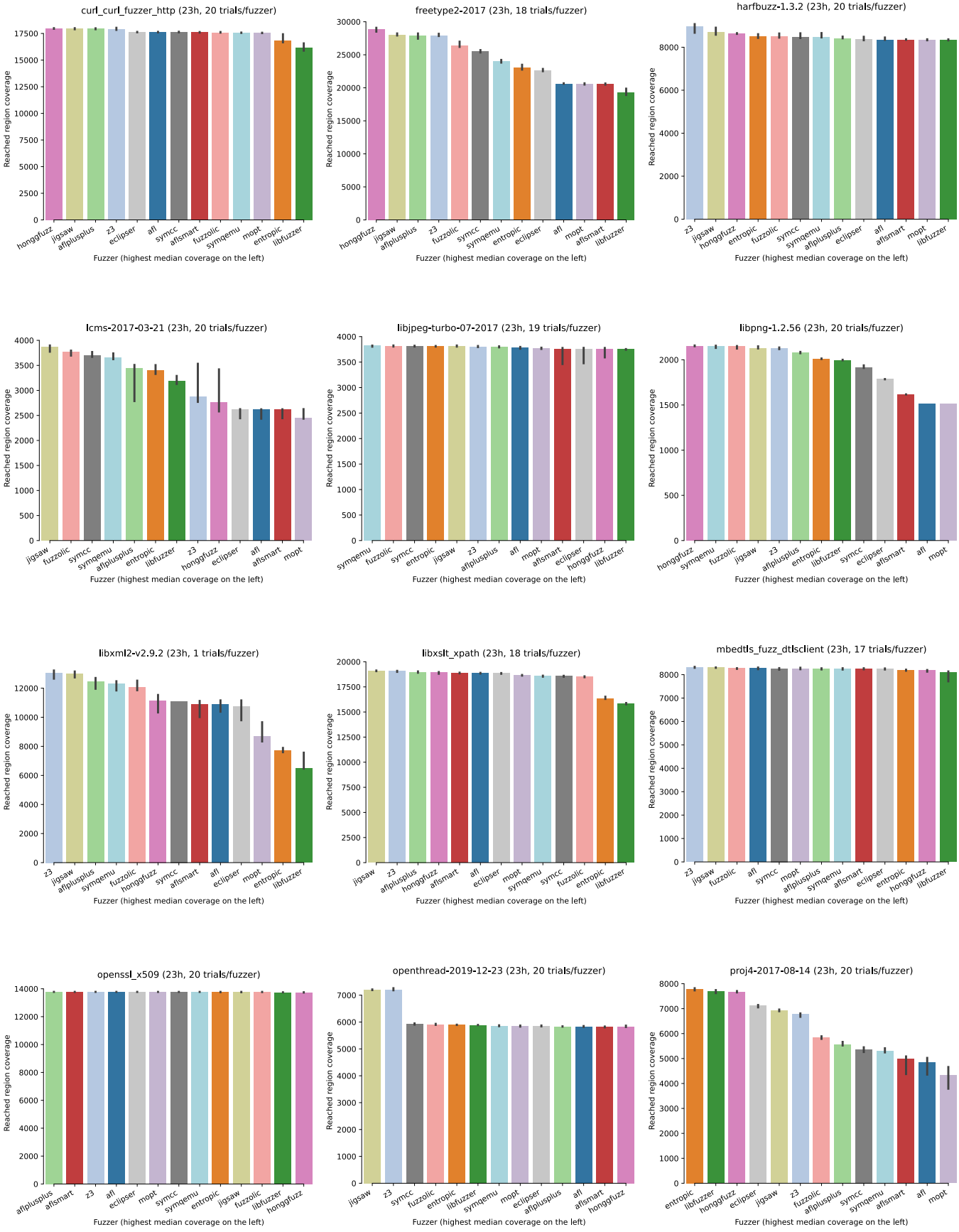
[77] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. Qsym: A practical concolic execution engine tailored for hybrid fuzzing. In *USENIX Security Symposium (Security)*, 2018.

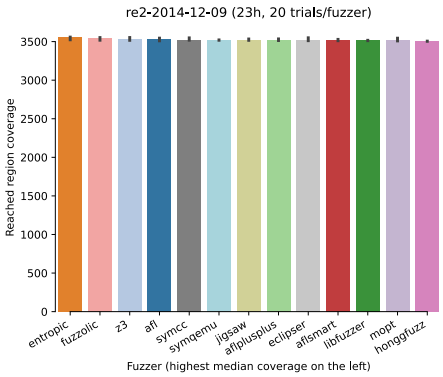
[78] Michal Zalewski. American fuzzy lop.(2014). http://lcamtuf.coredump.cx/afl, 2014.

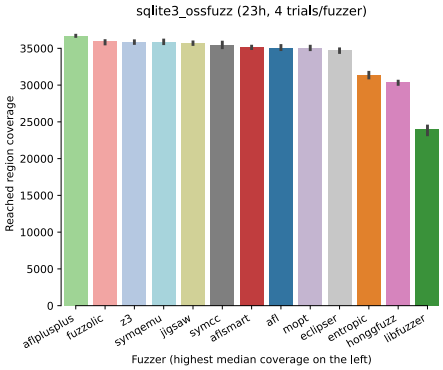
[79] Yaowen Zheng, Ali Davanian, Heng Yin, Chengyu Song, Hongsong Zhu, and Limin Sun. Firm-afl: high-throughput greybox fuzzing of iot firmware via augmented process emulation. In *USENIX Security Symposium (Security)*, 2019.

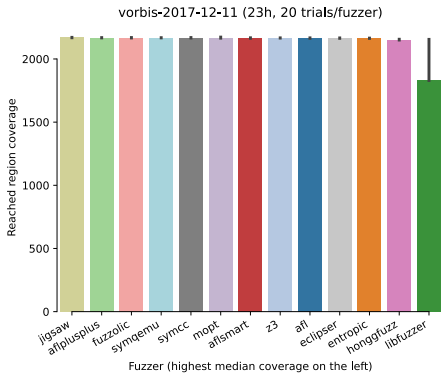
X. Appendix

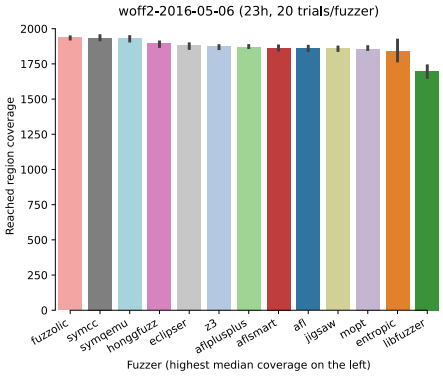
This section includes more results from the FuzzBench experiment. The full report can be retrieved at https://anonysp2022.github.io/.


curl_curl_fuzzer_http (23h, 20 trials/fuzzer)
freetype2-2017 (23h, 18 trials/fuzzer)
harfbuzz-1.3.2 (23h, 20 trials/fuzzer)
lcms-2017-03-21 (23h, 20 trials/fuzzer)
libjpeg-turbo-07-2017 (23h, 19 trials/fuzzer)
libpng-1.2.56 (23h, 20 trials/fuzzer)
libxml2-v2.9.2 (23h, 1 trials/fuzzer)
libxslt_xpath (23h, 18 trials/fuzzer)
mbedtls_fuzz_dtlsclient (23h, 17 trials/fuzzer)
openssl_x509 (23h, 20 trials/fuzzer)
openthread-2019-12-23 (23h, 20 trials/fuzzer)
proj4-2017-08-14 (23h, 20 trials/fuzzer)
Reached region coverage
Fuzzer (highest median coverage on the left)


re2-2014-12-09 (23h, 20 trials/fuzzer)
Reached region coverage
0
500
1000
1500
2000
2500
3000
3500
entropic
fuzzolic
z3
afl
symcc
symqemu
jigsaw
aflplusplus
eclipser
aflsmart
libfuzzer
mopt
honggfuzz
Fuzzer (highest median coverage on the left)


sqlite3_ossfuzz (23h, 4 trials/fuzzer)
Reached region coverage
0
5000
10000
15000
20000
25000
30000
35000
aflplusplus
fuzzolic
z3
symqemu
jigsaw
symcc
aflsmart
afl
mopt
eclipser
entropic
honggfuzz
libfuzzer
Fuzzer (highest median coverage on the left)


vorbis-2017-12-11 (23h, 20 trials/fuzzer)
Reached region coverage
0
500
1000
1500
2000
jigsaw
aflplusplus
fuzzolic
symqemu
symcc
mopt
aflsmart
z3
afl
eclipser
entropic
honggfuzz
libfuzzer
Fuzzer (highest median coverage on the left)


woff2-2016-05-06 (23h, 20 trials/fuzzer)
Reached region coverage
0
250
500
750
1000
1250
1500
1750
2000
fuzzolic
symcc
symqemu
honggfuzz
eclipser
z3
aflplusplus
aflsmart
afl
jigsaw
mopt
entropic
libfuzzer
Fuzzer (highest median coverage on the left)