SAVIOR: Towards Bug-Driven Hybrid Testing

Yaohui Chen,* Peng Li,† Jun Xu,‡ Shengjian Guo, † Rundong Zhou, † Yulong Zhang, † Tao Wei, † Long Lu,*

*Northeastern University †Baidu USA ‡Stevens Institute of Technology

***Abstract*—Hybrid testing combines fuzz testing and concolic execution. It leverages fuzz testing to test easy-to-reach code regions and uses concolic execution to explore code blocks guarded by complex branch conditions. As a result, hybrid testing is able to reach deeper into program state space than fuzz testing or concolic execution alone. Recently, hybrid testing has seen significant advancement. However, its code coverage-centric design is inefficient in vulnerability detection. First, it blindly selects seeds for concolic execution and aims to explore new code continuously. However, as statistics show, a large portion of the explored code is often bug-free. Therefore, giving equal attention to every part of the code during hybrid testing is a non-optimal strategy. It slows down the detection of real vulnerabilities by over 43%. Second, classic hybrid testing quickly moves on after reaching a chunk of code, rather than examining the hidden defects inside. It may frequently miss subtle vulnerabilities despite that it has already explored the vulnerable code paths.**

We propose SAVIOR**, a new hybrid testing framework pioneering a bug-driven principle. Unlike the existing hybrid testing tools,** SAVIOR **prioritizes the concolic execution of the seeds that are likely to uncover more vulnerabilities. Moreover,** SAVIOR **verifies all vulnerable program locations along the executing program path. By modeling faulty situations using SMT constraints,** SAVIOR **reasons the feasibility of vulnerabilities and generates concrete test cases as proofs. Our evaluation shows that the bug-driven approach outperforms mainstream automated testing techniques, including state-of-the-art hybrid testing systems driven by code coverage. On average,** SAVIOR **detects vulnerabilities 43.4% faster than** DRILLER **and 44.3% faster than** QSYM**, leading to the discovery of 88 and 76 more unique bugs, respectively. According to the evaluation on** `11` **well fuzzed benchmark programs, within the first 24 hours,** SAVIOR **triggers 481 UBSAN violations, among which 243 are real bugs.**

I. INTRODUCTION

Software inevitably contains defects [14, 64]. A large amount of these defects are security vulnerabilities that can be exploited for malicious purposes [54]. This type of vulnerable code has become a fundamental threat against software security. Contributed from both academia and industry, automated software testing techniques have gained remarkable advances in finding software vulnerabilities. In particular, people have widely used fuzz testing [2, 68] and concolic execution [51, 59] to disclose a great amount of vulnerabilities every year. Nevertheless, the inherent limitations of these two techniques impede their further applications. On one hand, fuzz testing quickly tests a program, but it hardly explores code regions guarded by complex conditions. On the other hand, concolic execution excels at solving path conditions but it frequently directs the execution into code branches containing a large number of execution paths (*e.g.,* loop). Due to these shortcomings, using fuzz testing or concolic execution alone often ends with large amounts of untested code after exhausting the time budget. To increase code coverage, recent works have experimented the idea of hybrid testing, which combines both fuzz testing and concolic execution [47, 66, 73].

The goal of hybrid testing is to utilize fuzzing in path exploration and leverage concolic execution to solve hard-to-resolve conditions. A hybrid approach typically lets fuzz testing run as much as possible. When the fuzzer barely makes any progress, the hybrid controller switches to the concolic executor which re-runs the generated seeds from fuzzing. During the run, the concolic executor checks each conditional branch to see whether its sibling branches remain untouched. If so, the concolic executor solves the constraints of the new branch and contributes a new seed for fuzzing. In general, this hybrid approach guides the fuzzer to new regions for deeper program space exploration.

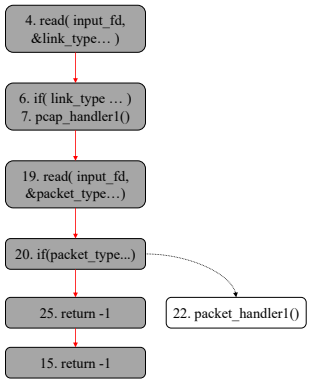
As shown in recent works [66, 73], hybrid testing creates new opportunities for higher code coverage. However, its coverage-driven principle unfortunately results in inefficiency when the end goal is vulnerability detection. Two key issues cause such inefficiency. First, existing approaches value all the seeds from fuzzing equally. However, the code regions reachable by a number of seeds might lack vulnerabilities and testing them is expensive (*e.g.,* constraint solving and extra fuzzing). Consequently, hybrid testing often exhausts the assigned time budget way before it finds any vulnerability. Second, hybrid testing could fail to identify a vulnerability even if it reaches the vulnerable code via the correct path. This is because hybrid testing primarily concentrates on covering the encountered code blocks in the manner of random exercise. This strategy oftentimes has low chances to satisfy the subtle conditions to reveal a vulnerability.

In this work, we design and implement SAVIOR (abbreviation for **S**peedy-**A**utomatic-**V**ulnerability-**I**ncentivized-**OR**acle), a hybrid, *bug-driven* testing method. To fulfill this goal, we use two novel techniques in SAVIOR:

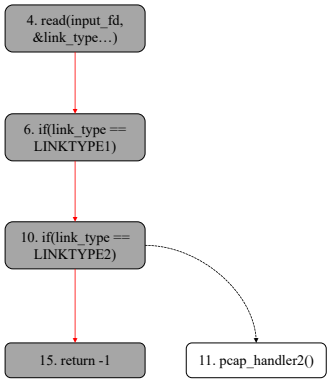
Bug-driven prioritization: Instead of running all seeds without distinction in concolic execution, SAVIOR prioritizes those that have higher possibilities of leading to vulnerabilities. Specifically, before the testing, SAVIOR analyzes the source code and statically labels the potentially vulnerable locations in the target program. Here SAVIOR follows existing methods [21, 35] to conservatively label all suspicious locations. Moreover, SAVIOR computes the set of basic blocks reachable from each branch. During dynamic testing, SAVIOR

```
1  int parse_pcap(){
2      int link_type;
3      /*read link-layer type from input*/
4      read(input_fd, &link_type, sizeof(int));
5      /*select a handler based on link_type*/
6      if(link_type == LINKTYPE1){
7      pcap_handler1();
8          return 0;
9      }
10     if(link_type == LINKTYPE2){
11         pcap_handler2();
12         return 0;
13     }
14     ...
15     return -1;
16 }
17 int pcap_handler1(){
18     int packet_type;
19     read(input_fd, &packet_type, sizeof(int));
20     if(packet_type == PACKET1){
21         packet_handler1();
22         return 0;
23     }
24     ...
25     return -1;
26 }
```

(a) A simplified version of the packet-parsing code in `tcpdump-4.9.2`, in which `pcap_handler2` contains vulnerabilities.



(b) The path followed by a seed that matches `LINKTYPE1` but mismatches `PACKET1`.



(c) The path followed by a seed that matches neither `LINKTYPE1` nor `LINKTYPE2`.

Fig. 1: A demonstrative example of hybrid testing. Figure 1a presents the code under test. Figure 1b and 1c are the paths followed by two seeds from the fuzzer. Their execution follows the red line and visits the grey boxes. Note that the white boxes connected by dotted lines are non-covered code.

prioritizes the concolic execution seeds that can visit more important branches (i.e., branches whose reachable code has more vulnerability labels). Intuitively, those branches may guard higher volumes of vulnerabilities and hence, prioritizing them could expedite the discovery of new vulnerabilities. As we will show in Section V, this prioritization enables SAVIOR to outperform DRILLER [66] and QSYM [73] with a **43.4%** and **44.3%** increase in bug discovering rate, respectively.

Bug-guided verification: Aside from accelerating vulnerability detection, SAVIOR also verifies the labeled vulnerabilities along the program path traversed by the concolic executor. Specifically, SAVIOR synthesizes the faulty constraint of triggering each vulnerability on the execution path. If such constraint under the current path condition is satisfiable, SAVIOR solves the constraint to construct a test input as the proof. Otherwise, SAVIOR proves that the vulnerability is infeasible on this path, regardless of the input. This SMT-solving based strategy, as demonstrated in Section V, enables DRILLER, QSYM, and SAVIOR to disclose not only all the listed bugs but also an additional group of bugs in LAVA-M [36]. Besides, it facilitates the three hybrid tools to find at least **22.2%**, **25%**, **4.5%** more UBSan violations.

This work is not the first one that applies hybrid testing to vulnerability detection. However, to the best of our knowledge, SAVIOR is the first work that explores bug-driven hybrid testing. On one hand, SAVIOR concentrates on software code that contains more potential vulnerabilities. This design not only brings faster coverage of vulnerabilities but also decreases the testing cost of the code that is less likely vulnerable. On the other hand, SAVIOR validates the vulnerabilities by the objective proofs of existence. In contrast, traditional hybrid testing methods can easily miss subtle cases. Moreover, the two proposed techniques are not limited to SAVIOR itself since they are general enough for other systematic software analysis methods. We will discuss the details in Section III. In summary, we make the following contributions.

- We design SAVIOR, a bug-driven hybrid testing technique. It substantially enhances hybrid testing with bug-driven prioritization and bug-guided verification.
- We build SAVIOR and show that our implementation can scale to a diverse set of real-world software.
- We demonstrate the effectiveness of SAVIOR by a comprehensive evaluation. In total, SAVIOR discovers **481** unique security violations in `11` well-studied benchmarks. On average, SAVIOR detects vulnerabilities **43.4%** faster than DRILLER and **44.3%** faster than QSYM, leading to the discovery of **88** and **76** more security violations in 24 hours.

The rest of this paper is organized as follows. Section II states the background of hybrid testing and motivates our research. Section III and Section IV present the design and implementation of SAVIOR in detail. Section V evaluates the core techniques of SAVIOR. Section VI summarizes the related work. Finally, we conclude this work in Section VII.

II. BACKGROUND AND MOTIVATION

This work is motivated by the limitations of hybrid testing in vulnerability detection. In this section, we first introduce the background of hybrid testing and then demonstrate the limitations by two examples.

A. Hybrid Testing

Hybrid testing combines fuzz testing and concolic execution to achieve high code coverage. For the ease of understanding, we use the example in Figure 1 to explain how it works. The explanation is based on Driller [66] since it has been the de facto implementation of hybrid testing.

The example in Figure 1 is taken from `tcpdump-4.9.2`. Figure 1a shows the code — it first uses the *link-layer* type from input to select a pcap handler and then uses the handler to dissect packets. Our objective is to test the entry function `parse_pcap` and reach the vulnerable function `pcap_handler2`.

In the test, we assume hybrid testing starts with a seed that executes the path shown in Figure 1b. After that, the fuzzer mutates the seed to run a second path shown in Figure 1c. It then, however, fails to synthesize inputs that match the *packet* type at line `20` and the *link-layer* type at line `10`, due to the huge mutation space (2^{32} possibilities). This situation prevents the fuzzer from testing the remaining code and makes hybrid testing switch to concolic execution.

After executing the seed that covers the path in Figure 1b, the concolic executor backtracks to the branch statement at line `20`. Solving the input `packet_type` to `PACKET1` by a SMT solver, the executor generates a new seed to cover that branch. Then, the hybrid controller suspends the concolic execution and resumes the fuzzer. Guided by the new seed, the fuzzer tests `packet_handler1` and switches back to concolic execution after that. This time, the concolic executor runs the seed, following the path in Figure 1c. After solving the branch condition at line `10`, it generates a seed for the flow from line `10` to line `11`. Further fuzz testing can finally reach the vulnerable code in `pcap_handler2`.

Note that the testing processes by different hybrid tools may vary from the above description. For instance, QSYM [73] keeps running concolic execution instead of invoking it in an interleaved manner. Despite those implementation differences, existing tools share a similar philosophy on scheduling the seeds to concolic execution. That is, they treat the seeds indiscriminately [66, 73], presumably assuming that these seeds have equal potentials in contributing to new coverage.

B. Motivation

Inefficiency in Covering Vulnerable Code: Although hybrid testing specializes in coverage-driven testing, it still needs substantial time to saturate hard-to-reach code compartments, which often overspends the time budget. To discover more vulnerabilities in a limited time frame, an intuitive way is to prioritize the testing of vulnerable code. However, the current hybrid testing method introduced in Section II-A does not meet this requirement.

Consider the example in Figure 1, where concolic execution chronologically runs the seeds to explore the paths shown in Figure 1b and Figure 1c. This sequence indeed postpones the testing of the vulnerable function `pcap_handler2`. The delay can be significant, because concolic execution runs slowly and the fuzz testing on `packet_handler1` may last a long time. In our experiments[1], DRILLER spends minutes on reaching `pcap_handler2` with the aforementioned schedule. However, if it performs concolic execution first on the path in Figure 1c, the time can reduce to seconds.

Not surprisingly, the delayed situations frequently happen in practice. As we will show in Section V, on average this defers DRILLER and QSYM to cover vulnerabilities by **43.4%** and **44.3%**, leading to reduced efficiency in vulnerability finding.

[1]SAVIOR is customized to do this test since DRILLER cannot run on `tcpdump`. More details can be found in Section V

```
1  static bfd_boolean load_specific_debug_section(enum
   ↪  dwarf_section_display_enum debug, asection
   ↪  *sec, void *file){
2
3      dwarf_section *section =
       ↪  &debug_displays[debug].section;
4
5      if (section->start != NULL){
6          if (streq (...)
7              return TRUE;
8          free (section->start);
9      }
10     ...
11     /*section->size is copied from input */
12     section->size = bfd_get_section_size (sec);
13
14     /*setting section->size as 0xffffffffffffffff
       ↪  on 64-bit systems or 0xffffffff on 32-bit
       ↪  systems, malloc will return a zero-byte
       ↪  buffer, leading to out of bound access */
15     section->start = malloc(section->size + 1);
16     ...
17 }
```

Fig. 2: A demonstrative example of limitation in finding defects by existing hybrid testing. This defect comes from `objdump-2.29` [15].

Deficiency in Vulnerability Detection: Hybrid testing often fails to identify a vulnerability even if it approaches the vulnerable location along the right path. Figure 2 demonstrates an integer overflow in `objdump-2.29`. At line `12`, the program copies a value from `sec` to `section→size`. Next, this value is used as the size of a memory allocation request at line `15`. By carefully handcrafting the input, an adversary can make `section→size` be the value $2^{32}-1$ on 32-bit systems or $2^{64}-1$ on 64-bit systems. This wraps `section→size+1` around to `0` and makes `malloc` return a zero-byte buffer. When the buffer is further used, a segfault or a memory leak would occur.

In this example, hybrid testing can quickly generate a seed to hit line `15`. However, it could barely trigger the integer overflow. As the program enforces no constraints on the input bytes that propagate to `section→size`, hybrid testing can only do random mutation to synthesize the extreme value(s). Taking into account the tremendous possibility space (2^{32} or 2^{64}), the mutation is unlikely to succeed.

III. DESIGN

A. Core Techniques

The design of SAVIOR is bug-driven, aiming to find bugs faster and more thoroughly. We propose two techniques to achieve the goal: *bug-driven prioritization* and *bug-guided verification*. Below we present an overview of our techniques.

Bug-driven prioritization: Recall that classic hybrid testing blindly schedules the seeds for concolic execution, without weighing their bug-detecting potentials. This can greatly defer the discovery of vulnerabilities. To remedy this limitation, SAVIOR collects information from the target source code to prioritize seeds which have higher potentials to trigger vulnerabilities. This approach, however, needs to predict the amount of vulnerabilities that running concolic execution on a seed could expose. The prediction essentially depends on two prerequisites: **R1** – *A method to assess the reachable code regions after the concolic execution on a seed* and **R2** – *A*

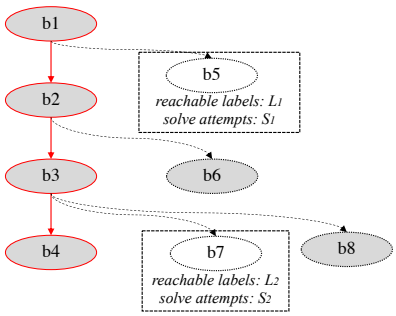


Fig. 3: An example showing how to estimate the bug-detecting potential of a seed. In this example, the seed follows the path `b1->b2->b3->b4`. Basic block `b5` and `b7` are unexplored and they can reach L_1 and L_2 UBSan labels, respectively. They have been attempted by constraint solving for S_1 and S_2 times. The final score for this seed is $\frac{e^{-0.05S_1} \times L_1 + e^{-0.05S_2} \times L_2}{2}$.

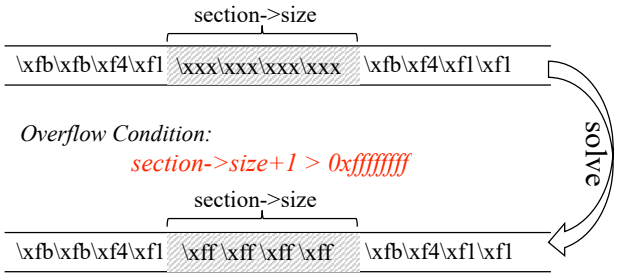


Fig. 4: Solving the integer overflow in Figure 2. This shows the case in a 32-bit system, but it applies to 64-bit as well.

metric to quantify the amount of vulnerabilities in a chunk of code. SAVIOR fulfills them as follows.

To meet **R1**, SAVIOR approximates the newly explorable code regions based on a combination of static and dynamic analysis. During compilation, SAVIOR statically computes the set of reachable basic blocks from each branch. At runtime, SAVIOR identifies the unexplored branches on the execution path of a seed and calculates the basic blocks that are reachable from those branches. We deem that these blocks become explorable code regions once the concolic executor runs that seed.

To meet **R2**, SAVIOR utilizes UBSan [21] to annotate three types of potential bugs (as shown in Table I) in the program under testing. It then calculates the UBSan labels in each code region as the quantitative metric for **R2**. As UBSan's conservative instrumentation may generate dummy labels, SAVIOR incorporates a static filter to safely remove useless labels. We discuss the details of this method in Section III-B1.

The above two solutions together ensure a sound analysis for identifying potential bugs. First, our static reachability analysis, as described in Section III-B1, is built upon a sound algorithm. It over-approximates all the code regions that may be reached from a branch. Moreover, UBSan adopts a conservative design, which counts all the operations that may lead to the undefined behavior issues listed in Table I [21, 35]. Facilitated by the two aspects of soundness, we can avoid mistakenly underrating the bug-detecting potential of a seed.

Following the two solutions, SAVIOR computes the importance score for each seed as follows. Given a seed with `n` unexplored branches $\{e_1, e_2, \ldots, e_n\}$, SAVIOR calculates the UBSan labels in the code that are reachable from these branches, respectively denoted as $\{L_1, L_2, \ldots, L_n\}$. Also note that, in the course of testing, SAVIOR has made $\{S_1, S_2, \ldots, S_n\}$ attempts to solve those branches. With these pieces of information, SAVIOR evaluates the importance score of this seed with a weighted average $\frac{1}{n} \times \sum_{i=1}^{n} e^{-0.05S_i} \times L_i$. L_i represents the potential of the i_{th} unexplored branch. We penalize L_i with $e^{-0.05S_i}$ to monotonically decrease its weight as the attempts to solve this branch grow. The rationale is that more failed attempts (usually from multiple paths) indicate a low success possibility on resolving the branch. Hence, we decrease its potential so that SAVIOR can gradually de-prioritize hard-to-solve branches. Lastly, SAVIOR takes the average score of each candidate branches in order to maximize the bug detection gain per unit of time. To better understand this scoring method, we show an example and explain the score calculation in Figure 3.

This scoring method is to ensure that SAVIOR always prioritizes seeds leading to more unverified bugs, while in the long run it would not trap into those with hard-to-solve branch conditions. First, it conservatively assesses a given seed by the results of sound reachability and bug labeling analysis. A seed which leads to more unexplored branches where more unverified bugs can be reached from will earn a higher score. Second, it takes into account runtime information to continuously improve the precision of the assessment. This online refinement is important because statically SAVIOR may hardly know whether a branch condition is satisfiable or not. Utilizing the history of constraint solving attempts, SAVIOR can decide whether a seemingly high-score branch is worth more resources in the future. As shown by our evaluation in Section V, this scoring scheme significantly accelerates the detection of UBSan violations, which empirically supports the effectiveness of our design.

Referring to our motivating example in Figure 1, the function `packet_handler1` has few UBSan labels while `pcap_handler2` contains hundreds of labels. Hence, the seed following Figure 1b has a lower score compared to the other seed which runs the path in Figure 1c. This guides SAVIOR to prioritize the latter seed, which can significantly expedite the exploration of vulnerable code.

Bug-guided verification: This technique also ensures a sound vulnerability detection on the explored paths that reach the vulnerable sites. Given a seed from fuzz testing, SAVIOR executes it and extracts the label of each vulnerability along the execution path. After that, SAVIOR verifies the predicates implanted in each label by checking the satisfiability under the current path condition — if the predicate is satisfiable then its corresponding vulnerability is valid. This enables SAVIOR to generate a proof of either vulnerability or non-existence along a specific program path. Note that in concolic execution, many new states with new branch constraints will be created.

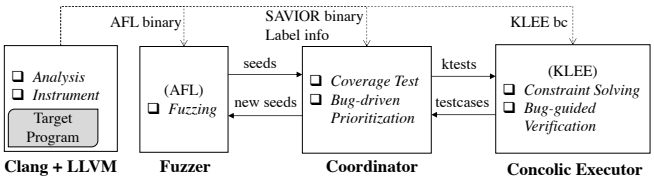


Fig. 5: System architecture of SAVIOR.

SAVIOR will prioritize the constraint solving for states who require bug-guided verification.

Going back to the example in Figure 2, classic hybrid testing misses the integer overflow at line `15`. In contrast, SAVIOR is able to identify it with bug-guided verification. Aided by the Clang sanitizer [21], SAVIOR instruments the potential overflows in a solver-friendly way (*i. e.,* the predicate of triggering this overflow is `section->size + 1 > 0xffffffff`). Due to the limited space, we present the instrumented IR code in Figure 10 at Appendix A. As demonstrated in Figure 4, following a seed to reach the integer overflow location, SAVIOR tracks that the value of `section->size` relies on a four-byte field in the input. By solving the vulnerability predicate, SAVIOR generates a witness value `0xffffffff` and triggers the vulnerability.

B. System Design

Figure 5 depicts the overall architecture of SAVIOR. It consists of a compiling tool-chain built upon Clang and LLVM, a fuzzer derived from AFL, a concolic executor ported from KLEE, and a hybrid coordinator responsible for the orchestration. We explain these components in details in the following sections.

1) The Compilation Tool-chain: SAVIOR's compilation tool-chain has multiple purposes including vulnerability labeling, control flow reachability analysis, and the targets building of different components.

Sound Vulnerability Labeling: In our design, we use Clang's Undefined Behavior Sanitizer (UBSan) [21] to label different families of potential bugs[2]. Table I summarizes those families used in SAVIOR and the operations pertaining to them.

We ignore other bug types listed in UBSan (*e.g.,* misaligned reference) since they are less likely to cause security issues. For each inserted label, we patch the Clang front-end to attach a `!saviorBugNum` metadata, aiding the reachability analysis that we will shortly discuss.

As explained in Section III-A, UBSan over-approximates the potential vulnerabilities. This approximation ensures soundness since it never misses true bugs. UBSan also models the conditional triggers of the labeled bugs as shown in Table I. E.g., *out-of-bound* (OOB) array access happens when the index `x` is not between *zero* and *array size minus 1*. At the time of bug-guided verification, SAVIOR solves each triggering condition to produce a witness of the bug or, prove that the bug never happens on current path in terms of the unsatisfiable

[2]Clang supports enabling checks on each individual bug family.

UB Families	***UBSan Labeling Details***	
	`Operation`	`Condition`
Out-of-bound array access	$array[x]$	$x < 0 \vee x \geq size(array)$
Oversized shift	$x \ll y, x \gg y$	$y < 0 \vee y \geq n$
Signed integer overflow	$x\ op_s\ y$	$x\ op_s\ y \notin [-2^{n-1}, 2^{n-1}-1]$
Unsigned integer overflow	$x\ op_u\ y$	$x\ op_u\ y > 2^n - 1$

TABLE I: Families of potential bugs that SAVIOR enables UBSan to label. Here, x, y are n-bit integers; $array$ is an array, the size of which is specified as $size(array)$; op_s and op_u refers to binary operators $+, -, \times, \div, \%$ over signed and unsigned integers, respectively.

condition. In Figure 10 at Appendix A, we present the IR with instrumented UBSan checks for the defect shown in Figure 2.

SAVIOR uses UBSan by default, while other labeling methods may also apply if they meet the following two properties. First, they can comprehensively annotate the potential vulnerabilities. Second, they can synthesize the triggering condition of each labeled vulnerability. Note that such condition must have data dependency on the program input. Otherwise, our concolic execution cannot correlate the input with the vulnerable conditions and hence, has no guidance for bug-guided verification. For instance, the AddressSanitizer [62] builds checks upon the status of its own red-zone, which is not applicable to SAVIOR at the moment.

UBSan's conservative approximation inevitably introduces false positives and might mislead SAVIOR's prioritization. In practice, we incorporate a static counter-measure to reduce fake labels. Specifically, we trim a label when all the following requirements hold: *1)* The label's parent (basic block) is its immediate dominator [65]; *2)* The IR variables involved in the vulnerability conditions are not re-defined between the label and its parent; *3)* The parent basic block has constraints that conflict with the vulnerability conditions, and these constraints are enforced by constant values. The first two points ensure that the constraints added by the parent will persist upon reaching the label, and the third point indicates that the conflict always arises, regardless of the input and the execution path. Therefore, we can safely remove this label.

```
1  char array[MAX]; // 0 < MAX < INT_MAX
2  for(int i = 0; i < MAX;){
3     array[i] = getchar();//LABEL: OOB access
4     i++;//LABEL: integer-overflow
5  }
```

For instance, the code above has two labels that meet the three requirements. In this example, the variable `i` ranges from `0` to `MAX`, meaning that neither the array access at line `3` can be *out-of-bound* nor the self increment at line `4` can cause an integer overflow. SAVIOR hence removes the two labels. In Table IX at Appendix A, we summarize the number of labels that are removed from each of our benchmark programs. On average, we can conservatively reduce 5.36% of the labels.

Reachability Analysis: This analysis counts the number of vulnerability labels that can be forwardly reached by each basic block in the program control flow graph (CFG). It proceeds with two phases. The first step constructs an inter-

procedure CFG. The construction algorithm is close to the method implemented in SVF [67]. It individually builds intra-procedure CFGs for each function and then bridges function-level CFGs by the caller-callee relation. To resolve indirect calls, our algorithm iteratively performs Andersen's point-to analysis and expands the targets of the calls. This prevents SAVIOR from discarding aliasing information of indirect calls and therefore, our prioritization would not miscount the number of vulnerability labels. By examining the CFGs, we also extract the edge relations between a basic block and its children for further use in the hybrid coordinator.

The second step is to calculate the UBSan labels that are reachable from each basic block in the constructed inter-procedure CFG. Specifically, we identify the regions of code that a basic block can reach and count the number of UBSan labels in those regions. In SAVIOR, we deem this number as the importance metric of that basic block and use it for bug-driven prioritization. For example, in Figure 6 the basic block **BB** can reach 8 other basic blocks while 3 of them have *UBSan* labels. Thereby we output 3 as the number of reachable UBSan labels for **BB**. Note that each basic block at most has one label after Clang's compilation.

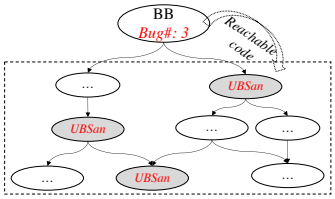


Fig. 6: A demonstrative example of reachability analysis. The target **BB** can "reach" 3 UBSan labels.

Target Building: After the labeling and the reachability analysis, SAVIOR's compiling tool-chain begins its building process. It compiles three binaries from the source code — a fuzzing-binary for the fuzzer, a SAVIOR-binary for the coordinator, and a LLVM bitcode file for the concolic executor. In particular, the SAVIOR-binary is instrumented to print the unique IDs of the executed basic blocks. With this design, SAVIOR completely decouples the fuzzer, the concolic executor and the coordinator, thus it supports quick replacement of any components.

2) The Coordinator: The coordinator bridges the fuzzer and the concolic executor. It keeps polling seeds from the fuzzer's queue and prioritizes those with higher importance for concolic execution. We explain the details as follows.

Bug-driven Prioritization: In a polling round, the coordinator operates the new seeds in the fuzzer's queue after last round. Each seed is fed to the SAVIOR-binary and the coordinator updates two pieces of information based on the execution result. First, it updates the global coverage information. The coverage computation here follows AFL's original approach. That is, we take the hit counts of an edge in the following ranges as different coverage: $[1], [2], [3], [4, 7], [8, 15], [16, 31], [32, 127], [128, \infty)$. Second, the coordinator records the sequence of basic blocks

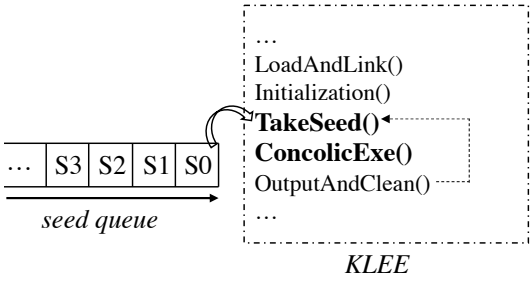


Fig. 7: Fork server mode in KLEE. In this mode, KLEE only performs initialization once and reuses the same executor for all the received seeds.

visited by each seed. Using the updated coverage information, the coordinator assigns a score to each seed following the scheme presented in Section III-A. Here, we re-score all the seeds except those already tested by our concolic executor, since the coverage information is dynamically adjusted.

Finally, the coordinator selected the top-ranked seeds and feed them into the input queue of the concolic executor. If two seeds have the same score, the coordinator prefers the seed with the `+cov` property. `+cov` indicates that the seed brings new code coverage.

Post-processing of Concolic Execution: Going beyond seed scheduling for concolic execution, the coordinator also need to triage the new seeds generated by the concolic executor for the fuzzer. First, it re-runs the new seeds and retains those who provide new coverage or can reach uncovered bug labels. As a result, SAVIOR transfers the valuable test cases from the concolic executor to the fuzzer.

Second, the coordinator updates the number of solving attempts upon uncovered branches. If a branch remains uncovered, its solving attempts would be increased by 1. As such, a branch having a much higher solving attempt value will be de-prioritized.

3) The Concolic Executor: The concolic executor replays the seeds scheduled by the coordinator and chooses to solve branch conditions based on coverage information. In addition, it also performs bug-guided verification.

Independent Coverage Scheme: When encountering a branch instruction the concolic executor needs to decide whether to solve this branch's condition. An intuitive design is to reuse the coverage information from the coordinator. However, since our coverage scheme is ID based, yet as KLEE invokes a group of transformations on the target bitcode, this leads to numerous mismatches between the edge IDs in the SAVIOR-binary and the KLEE bitcode. To tackle this problem, we opt to use KLEE's internal coverage information to better decouple the concolic executor and other components.

Fork Server Mode: Before running a seed, KLEE needs to perform a group of initialization, including bitcode loading, library bitcode linking, and global data preparation to place the program under testing into the virtual machine. This

initialization process, however, typically takes a long time on large bitcode files. For instance, the initialization time for `tcpdump` is usually several times longer than the actual concolic execution time. To address this issue, we introduce an optimization named *fork server* mode for the KLEE concolic executor (as shown in Figure 7). Technical details are explained in Section IV.

Bug-guided Verification: Our concolic executor also performs bug-guided verification. Once an non-covered vulnerability label is reached, we endeavor to solve the triggering constraint following the current path. If the solving succeeds, KLEE generates a seed as the proof of the vulnerability.

In certain cases, the path constraints may conflict with the vulnerability triggering conditions, while that vulnerability can indeed happen following the same path (with fewer constraints). QSYM [73] summarizes this issue as the over-constraint problem. We adopt QSYM's optimistic solving strategy only on solving the vulnerability conditions. However, the relaxed-constraint may also produce a false positive, and we do not count a vulnerable label as being covered through relaxed-constraint solving.

Timeout on Concolic Execution: To prevent the concolic execution from hanging on localized code regions (*e.g.,* , deep loops and blocking IO), the concolic executor usually needs a time threshold while running a seed. QSYM adjusts this timing budget by watching AFL's status. If the number of hanging seeds increases, QSYM increases the timeout (up to `10` minutes). We set the timeout to be proportional to the number of uncovered branches that a seed can reach. The rationale is that those seeds need more time for constraint solving and such setting benefits higher bug coverage.

IV. Implementation

We have implemented SAVIOR, which can be applied to software as sophisticated as Baidu's Apollo Autonomous Driving System [5, 37]. SAVIOR consists of four major components: a compiling tool-chain built on top of Clang and LLVM-4.0, a fuzzing component based on AFL-2.5b [2], a concolic executor built atop KLEE [27] (with LLVM-3.6), and a python middle-ware which coordinates the fuzzing component and the concolic executor. In total, our implementation has about 3.3K lines of python code and 4K lines of C/C++ code. SAVIOR can run on both 32-bit and 64-bit systems, and it can support both 32-bit and 64-bit targets. In the following, we discuss the important implementation details.

Concolic Executor: We develop our concolic executor based on KLEE-3.6. The original KLEE aims at full symbolic execution, and it does not support concolic execution. We port a concolic executor from KLEE's symbolic executor. Specifically, the concolic executor attaches the concrete input as the `assignment` property in the initial state. It then symbolically interprets each instruction as KLEE originally does. On reaching a conditional statement, it always follows the branch that matches the concrete input. For the other branch, if not covered, the concolic executor solves the conditions and generate a corresponding testcase. The state following that branch is then immediately terminated. When generating the seed, our concolic executor copies the un-constrained bytes from the input, instead of padding with random values.

Another limitation of KLEE is that the initialization phase is notoriously time-consuming. To overcome this, we introduce a fork server mode. In a run, KLEE first sets up the environments with bitcode loading, library linking, and preparing for globals and constants. These are then followed by the initialization of an `Executor`. By default, the `Executor` executes one seed and then destructs itself. In our implementation, after the execution of one seed, we clean up any stateful changes introduced in last execution (including destructing the memory manager, clearing the global data objects, and erasing all the remaining states). Then we reuse the `Executor` to run a new seed from the input queue. In this mode, we avoid repeating the lengthy environments setup.

Recall that we invoke UBSan to label potentially vulnerable operations. At the IR level, UBSan replaces those operations with LLVM intrinsic functions, which are incomprehensible by KLEE. We replace those intrinsic functions with general LLVM IR so that KLEE can execute without exceptions. The replacements follow those that KLEE already enforced [10].

By default, KLEE redirects un-modeled external functions (*e.g.,* system calls) to the native code. This causes two issues. First, KLEE is unaware of their effects on the symbolic address space, which can interrupt memory operations. For instance, the function `strdup` allocates a new buffer and copies data from the source to this buffer. However, KLEE cannot capture this allocation due to the lack of modeling. On future accesses to this buffer, KLEE will throw an out-of-bound access error. There are many similar cases, such as `getenv`. We extend KLEE's environment model to include the symbolic versions of those functions. Second, KLEE concretizes the data passed to the external functions and adds constant constraints on such data for future execution. However, this may over-constraint the concretized variables. For instance, KLEE concretizes the data written to standard output or files. This leads to over-constraints – When the concretized data is later used in constraint solving, KLEE will not be able to find a satisfying solution. To address this issue, we prevent KLEE from adding constraints on concretization. This scheme, following the design of S2E [31] and QSYM [73], ensures that we never miss solutions for non-covered branches.

Last but not least, stock KLEE provides limited support for software written in `C++`. Since a lot of the C++ programs rely on the standard `C++` library (*e.g.,* libstdc++ on Linux) but KLEE neither models this library nor supports the semantics of calls to this library. Therefore, KLEE frequently aborts the execution in the early stage of running a `C++` program. We customize the GNU libstdc++ library to make it compilable and linkable to KLEE. Considering that many libstdc++ functions also access in-existent devices (*e.g.,* Random), we also build models of those devices.

Fuzzers	*Setup*		
	Source	Instances	Note
AFL	[2]	1 AFL master; 2 AFL slaves	N/A
AFLGo	[1]	1 AFLGo master; 2 AFLGo slaves	Use in-lined `lava_get` as target locations of guided fuzzing
TFuzz	[19]	3 AFL jobs (adjust default argument to Fuzzer)	Use the docker environment prepared at [19] for evaluation
Angora	[3]	3 Angora threads (with option `"-j 3"`)	Patch Lava to support Angora, as suggested by the developers [18]
Driller	Self-developed	1 concolic executor; 1 AFL master; 1 AFL slave	Follow the original `Driller` in scheduling concolic execution [7]
QSYM	[17]	1 concolic executor; 1 AFL master; 1 AFL slave	N/A
SAVIOR	Self-developed	1 concolic executor; 1 AFL master; 1 AFL slave	Use in-lined `lava_get` as labels of vulnerabilities

TABLE II: Fuzzer specific settings in evaluation with Lava-M.

V. Evaluation

SAVIOR approaches bug-driven hybrid testing with the key techniques of bug-driven prioritization and bug-guided verification. In this section, we evaluate these techniques and our evaluation centers around two questions:

- *With bug-driven prioritization, can hybrid testing find vulnerabilities quicker?*
- *With bug-guided verification, can hybrid testing find vulnerabilities more thoroughly?*

To support our evaluation goals, we prepare two groups of widely-used benchmarks. The first group is the LAVA-M data-set [36]. This data-set comes with artificial vulnerabilities, and the ground truth is provided. The second group includes a set of `8` real-world programs. Details about these programs are summarized in Table V. All these programs have been extensively tested in both industry [16] and academia [57, 66, 73]. In addition, they represent a higher level of diversity in functionality and complexity.

Using the two benchmarks, we compare SAVIOR with the most effective tools from related families. To be specific, we take AFL [2] as the baseline of coverage-based testing. As SAVIOR performs testing in a directed manner, we also include the state-of-the-art directed fuzzer, AFLGo [25]. To handle complex conditions, recent fuzzing research introduces a group of new techniques to improve code coverage. From this category, we cover TFuzz [56] and Angora [29], because they are open-sourced and representatives of the state-of-the-art. Finally, we also consider the existing implementations of hybrid testing, Driller [66] and QSYM [73].

Note that the original Driller has problems of running many of our benchmarks, due to lack of system-call modeling or failure to generate test cases (even with the patch [6] to support input from files). This aligns with the observations in [73]. In the evaluation, we re-implement Driller on the top of SAVIOR. More specifically, it runs AFL as the fuzzing component and it invokes the concolic executor once the `pending_favs` attribute in AFL drops to 0. These implementations strictly follow the original Driller [7]. Similar to the Angr-based concolic executor in Driller, our KLEE-based concolic executor focuses on generating new seeds to cover untouched branches. In addition, we keep the relaxed constraint solving and the fork-server mode. These two features increase the effectiveness and efficiency of Driller without introducing algorithmic changes.

In the following, we will explain the experimental setups and evaluation results for the two groups of benchmarks.

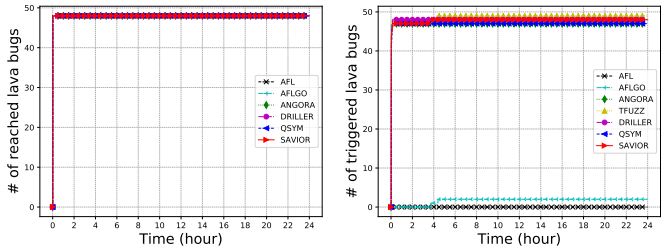
A. Evaluation with LAVA-M

1) Experimental Setup: In this evaluation, we run each of the fuzzers in Table II with the four LAVA-M programs and we use the seeds shipped with the benchmark. For consistency, we conduct all the experiments on Amazon EC2 instances (Intel Xeon E5 Broadwell 64 cores, 256GB RAM, and running Ubuntu 16.04 LTS), and we sequentially run all the experiments to avoid interference. In addition, we assign each fuzzer 3 free CPU cores to ensure fairness in terms of computation resources. Each test is run for 24 hours. To minimize the effect of randomness in fuzzing, we repeat each test 5 times and report the average results.

In Table II, we also summarize the settings specific to each fuzzer, including how we distribute the 3 CPU cores and the actions we take to accommodate those fuzzers. In LAVA-M, each artificial vulnerability is enclosed and checked in a call to `lava_get` (in-lined in our evaluation). We use these calls as the targets to guide AFLGo and we mark them as vulnerability labels to enable bug-driven prioritization in SAVIOR. In addition, as the vulnerability condition is hard-coded in the `lava_get` function, we naturally have support for bug-guided verification. Finally, for Angora, we adopt the patches as suggested by the developers [18].

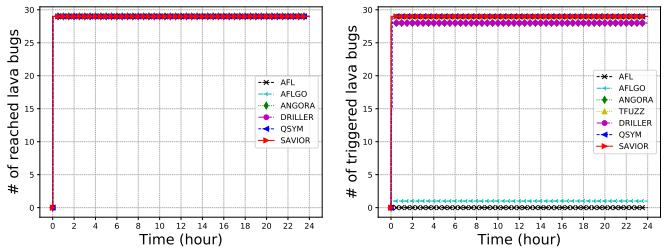
2) Evaluation Results: In the left column of Figure 8, we show how many vulnerabilities are reached over time by different fuzzers. The results demonstrate that all the fuzzers can instantly cover the code with LAVA vulnerabilities. However, as presented in the right column of Figure 8, TFuzz, Angora, Driller, QSYM, and SAVIOR are able to trigger most (or all) of the vulnerabilities while AFL and AFLGo can trigger few. The reason behind is that the triggering conditions of LAVA vulnerabilities are all in the form of 32-bit magic number matching. Mutation-based fuzzers, including AFL and AFLGo, can hardly satisfy those conditions while the other fuzzers are all featured with techniques to solve them.

Vulnerability Finding Efficiency: Despite TFuzz, Angora, Driller, QSYM, and SAVIOR all trigger large numbers of LAVA vulnerabilities, they differ in terms of efficiency. TFuzz quickly covers the listed vulnerabilities in `base64` and `uniq`. This is attributable to that (1) TFuzz can reach all the vulnerabilities with several initial seeds and (2) TFuzz can transform the program to immediately trigger the encountered vulnerabilities. Note that we do not show the results of TFuzz on `md5sum` and `who`, because TFuzz gets interrupted



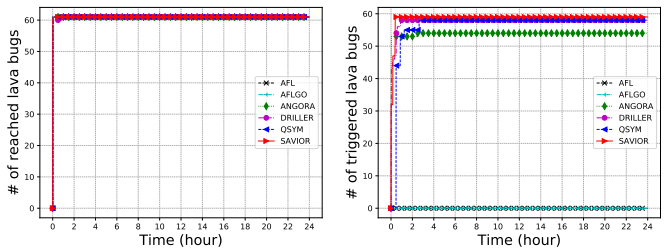
(a) Number of bugs reached in base64

(b) Number of bugs triggered in base64



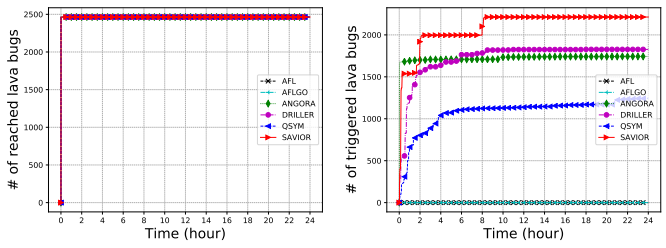
(c) Number of bugs reached in uniq

(d) Number of bugs triggered in uniq



(e) Number of bugs reached in md5sum

(f) Number of bugs triggered in md5sum



(g) Number of bugs reached in who

(h) Number of bugs triggered in who

Fig. 8: Evaluation results with LAVA-M. The left column shows the number of Lava bugs reached by different fuzzers and the right column shows the number of LAVA bugs triggered by the fuzzers. For TFUZZ, we only present the number of triggered bugs in `base64` and `uniq`, as the other results are not reliable due to a broken third-party dependency.

because of a broken dependency [3]. For all the cases, ANGORA triggers the vulnerabilities immediately after its start. The main reason is that the "black-box function" pertaining to all LAVA vulnerabilities is `f(x) = x` and the triggering conditions are like `f(x) == CONSTANT`. ANGORA always starts evaluating such functions with `x = CONSTANT` and hence, it can instantly generate seeds that satisfy the vulnerability conditions. In the case of `who`, ANGORA does not find all the vulnerabilities because of its incomplete dynamic taint analysis.

[3]The broken component is the QEMU based tracer in Angr [4]. This has been confirmed with the developers.

Fuzzers	*Fuzzing results*			
	`base64`	`uniq`	`md5sum`	`who`
AFL	0 (0%)	0 (0%)	0 (0%)	0 (0%)
AFLGO	2 (5%)	1 (4%)	0 (0%)	0 (0%)
TFUZZ	47 (100%)	29 (100%)	N/A	N/A
ANGORA	47 (100%)	28 (100%)	54 (95%)	1743 (79%)
DRILLER	48 (100%)	28 (100%)	58 (100%)	1827 (78%)
QSYM	47 (100%)	29 (100%)	58 (100%)	1244 (53%)
SAVIOR	48 (100%)	29 (100%)	59 (100%)	2213 (92%)
Listed	**44**	**28**	**57**	**2136**

TABLE III: LAVA-M Bugs triggered by different fuzzers (before bug-guided verification). "X%" indicates that X% of the listed LAVA bugs are triggered.

Fuzzers	*Fuzzing results*			
	`base64`	`uniq`	`md5sum`	`who`
AFL	48 (100%)	29 (100%)	59 (100%)	2357 (96.3%)
AFLGO	48 (100%)	29 (100%)	59 (100%)	2357 (96.3%)
TFUZZ	47 (100%)	29 (100%)	N/A	N/A
ANGORA	48 (100%)	29 (100%)	59 (100%)	2357 (96.3%)
DRILLER	48 (100%)	29 (100%)	59 (100%)	2357 (96.3%)
QSYM	48 (100%)	29 (100%)	59 (100%)	2357 (96.3%)
SAVIOR	48 (100%)	29 (100%)	59 (100%)	2357 (96.3%)
Listed	**44**	**28**	**57**	**2136**

TABLE IV: LAVA-M Bugs triggered by different fuzzers (after bug-guided verification). "X%" indicates that X% of the listed LAVA bugs are triggered.

Regarding the three hybrid tools, they trigger every vulnerability that their concolic executors encounter. In the cases of `base64`, `uniq`, and `md5sum`, their concolic executors can reach all the vulnerabilities with initial seeds. This explains why they all quickly trigger the listed vulnerabilities, regardless of their seed scheduling.

In the case of `who`, even though the fuzzing component quickly generates seeds to cover the vulnerable code, the concolic executor takes much longer to run those seeds. For instance, while executing the inputs from AFL, QSYM needs over 72 hours of continuous concolic execution to reach all the LAVA bugs in `who`. Differing from DRILLER and QSYM, SAVIOR prioritizes seeds that have a higher potential of leading to Lava bugs. As demonstrated by the results of `who` in Table III, our technique of *bug-driven prioritization indeed advances the exploration of code with more vulnerabilities*. Note that DRILLER (with a random seed scheduling) moves faster than QSYM. This is because QSYM prioritizes concolic execution on small seeds, while reaching the vulnerabilities in `who` needs seeds with a larger size.

Vulnerability Finding Thoroughness: We further evaluate our bug-guided verification design. Specifically, we run the seeds generated by all the fuzzers with our concolic executor. In this experiment, we only perform constraint solving when a vulnerability condition is encountered. As shown in Table IV, bug-guided verification facilitates all the fuzzers to not only cover the listed LAVA bugs but also disclose an extra group of Lava bugs. Due to limited space, those additionally identified bugs are summarized in Table X at Appendix. Such results *strongly demonstrate the promising potential of bug-guided verification to benefit fuzzing tools in vulnerability findings*.

Programs				*Settings*	
Name	Version	Driver	Source	Seeds	Options
libpcap	4.9.2/1.9.0	tcpdump	[20]	build-in	-r @@
libtiff	4.0.10	tiff2ps	[12]	AFL	@@
libtiff	4.0.10	tiff2pdf	[12]	AFL	@@
binutils	2.31	objdump	[8]	AFL	-D @@
binutils	2.31	readelf	[8]	AFL	-A @@
libxml2	2.9.7	xmllint	[13]	AFL	@@
libjpeg	9c	djpeg	[11]	AFL	
jasper	master	jasper	[9]	AFL	-f @@ -T pnm

TABLE V: Real-world benchmark programs and evaluation settings. In the column for Seeds, AFL indicates we reuse the testcases provided in AFL and build-in indicates that we reuse the test cases shipped with the program.

B. Evaluation with Real-world Programs

1) Experimental Setup: In this evaluation, we prepare 8 programs. Details about these programs and the test settings are summarized in Table V. All the programs have been extensively tested by both industry [16] and academic researches [57, 66, 73]. Since different seed inputs and execution options could lead to varying fuzzing results [49, 58], we follow existing works to use the seeds shipping with AFL or the vendors, as well as to configure the fuzzing options. Similar to our evaluation with LAVA-M, we conduct all the experiments on Amazon EC2 instances. To reduce randomness during testing, we run each test 5 times and report the average results. In addition, we leverage Mann Whitney U-test [53] to measure the significance of our improvements, following the suggestion by George etc [49].

In this evaluation, we also prepare the setups that are specific to each fuzzing tool. These setups mostly follow Table II except the following. First, we use UBSan labels as the target locations for AFLGO and as the guidance of bug-driven prioritization in SAVIOR. Second, to prevent ANGORA from terminating the fuzzing process once it encounters un-instrumented library functions, we follow suggestions from the developers and add the list of un-instrumented functions into ANGORA's dfsan_abilist.txt configuration file. Third, we do not include TFUZZ, because it does not function correctly on our benchmark programs due to issues in the aforementioned third-party component. Furthermore, we prepare these benchmark programs such that they are instrumented with UBSan for all fuzzers to ensure a fair comparison. This also means that bug-guided verification is enabled by default in DRILLER, QSYM, and SAVIOR.

2) Evaluation Results: In Figure 9, we summarize the results of our second experiment. It shows the outputs over time from two metrics, including the number of triggered UBSan bugs and basic block coverage. In addition, we calculate the p-values for Mann Whitney U-test of SAVIOR vs. DRILLER and SAVIOR vs. QSYM. Note that we use the IDs of UBSan labels for de-duplication while counting the UBSan bugs, as each UBSan label is associated with a unique potential defect. In the following, we delve into the details and explain how these results testify our design hypotheses.

Vulnerability Finding Efficiency: As shown in Figure 9 (the left column of each program), SAVIOR triggers UBSan violations with a pace generally faster than all the other fuzzers. In particular, it outperforms DRILLER and QSYM in all the cases except djpeg. On average, SAVIOR discovers vulnerabilities **43.4%** faster than DRILLER and **44.3%** faster than QSYM. The low p-values (< 0.05)[4] of Mann Whitney U-test well support that these improvements are statistically significant. Since the three hybrid tools only differ in the way of seed scheduling, these results strongly demonstrate that *the scheduling scheme in* SAVIOR*— bug-driven prioritization — accelerates vulnerability finding*. In the case of djpeg, all six fuzzers trigger the same group of UBSan violations. This is because djpeg has a tiny code base, with which these fuzzers quickly saturate on code exploration. In addition, the conditions of those UBSan violations are simple that even mutation-based approaches can solve. For a better reference, we also summarize the number of triggered violations at the end of 24 hours in Table XII at Appendix A-D.

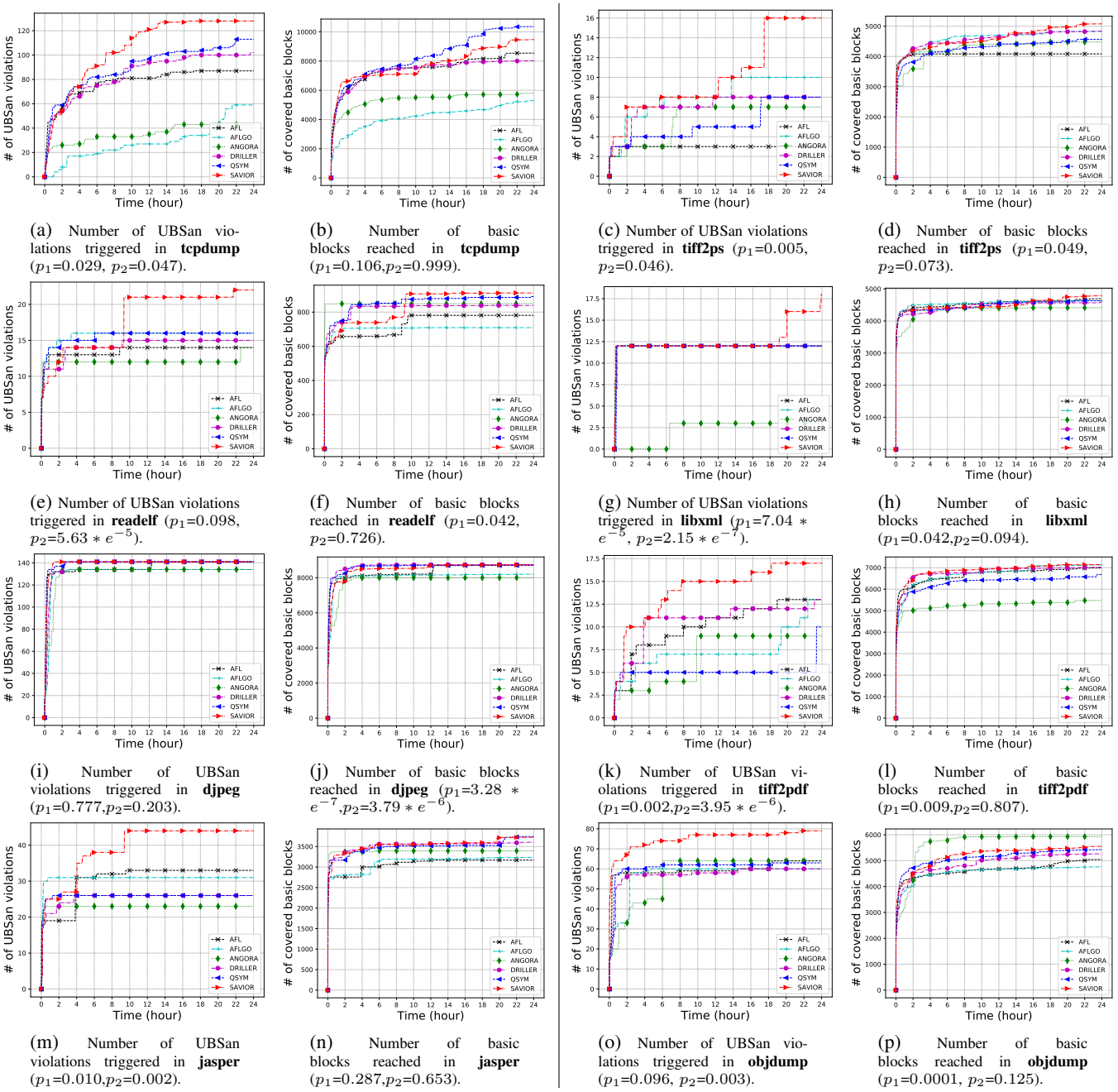
Going beyond, we examine the number of labels that are reached by different fuzzers. In Table VI, we list the average results from our 24-hour tests. Not surprisingly, the hybrid tools cover higher volumes of UBSan labels than the ordinary fuzzers. This is likely because a hybrid tool can solve complex conditions, enabling the coverage on the code and labels behind. Among the hybrid tools, SAVIOR reaches **19.68%** and **15.18%** more labels than DRILLER and QSYM, respectively. Such results are consistent with the number of triggered UBSan violations. This also signifies that our bug-driven prioritization guides SAVIOR to spend more resources on code with richer UBSan labels. In the case of djpeg, SAVIOR nearly ties with the other tools. This is due to a similar reason as explained above.

We further find that the efficiency boost of SAVIOR in vulnerability finding is not due to high code coverage. As shown in Figure 9 (the right column for each program), we compare the code coverage of the six fuzzers. As demonstrated by the results, the efficiency of code coverage and UBSan violation discovery are not positively correlated. Particularly, in the case of tcpdump, libxml, tiff2pdf, objdump and jasper, SAVIOR covers code in a similar or even slower pace than DRILLER and QSYM (the high p-values also support that SAVIOR is not quicker). However, SAVIOR triggers UBSan violations significantly quicker in these cases. Such results validate the above hypothesis with high confidence.

Vulnerability Finding Thoroughness: In this experiment, we also measure the performance of bug-guided verification in enhancing the thoroughness of vulnerability finding. Specifically, we re-run the seeds from all the fuzzers with our concolic executor. In this test, we enable SAVIOR to do constraint solving only when encountering un-solved UBSan labels.

In Table VII, we summarize the comparison results. For all the 8 programs, bug-guided verification facilitates different

[4]The p-values of readelf and objdump are larger than 0.05 but they are at the level of quasi-significance. In the two programs, the variances are mainly due to randomness.



(a) Number of UBSan violations triggered in **tcpdump** (p_1=0.029, p_2=0.047).

(b) Number of basic blocks reached in **tcpdump** (p_1=0.106,p_2=0.999).

(c) Number of UBSan violations triggered in **tiff2ps** (p_1=0.005, p_2=0.046).

(d) Number of basic blocks reached in **tiff2ps** (p_1=0.049, p_2=0.073).

(e) Number of UBSan violations triggered in **readelf** (p_1=0.098, p_2=$5.63 * e^{-5}$).

(f) Number of basic blocks reached in **readelf** (p_1=0.042, p_2=0.726).

(g) Number of UBSan violations triggered in **libxml** (p_1=$7.04 * e^{-5}$, p_2=$2.15 * e^{-7}$).

(h) Number of basic blocks reached in **libxml** (p_1=0.042,p_2=0.094).

(i) Number of UBSan violations triggered in **djpeg** (p_1=0.777,p_2=0.203).

(j) Number of basic blocks reached in **djpeg** (p_1=$3.28 * e^{-7}$,p_2=$3.79 * e^{-6}$).

(k) Number of UBSan violations triggered in **tiff2pdf** (p_1=0.002,p_2=$3.95 * e^{-6}$).

(l) Number of basic blocks reached in **tiff2pdf** (p_1=0.009,p_2=0.807).

(m) Number of UBSan violations triggered in **jasper** (p_1=0.010,p_2=0.002).

(n) Number of basic blocks reached in **jasper** (p_1=0.287,p_2=0.653).

(o) Number of UBSan violations triggered in **objdump** (p_1=0.096, p_2=0.003).

(p) Number of basic blocks reached in **objdump** (p_1=0.0001, p_2=0.125).

Fig. 9: Evaluation results with real-world programs. Each program takes two columns, respectively showing the number of triggered UBSan violations and the amount of covered basic blocks by the fuzzers over 24 hours. p_1 and p_2 are the p-values for the Mann Whitney U-test of SAVIOR vs. DRILLER and SAVIOR vs. QSYM, respectively.

fuzzers to trigger new violations. The average increase ranges from **4.5%** (SAVIOR) to **61.2%** (ANGORA). In particular, it aids ANGORA to trigger 82 new UBSan bugs in total. In the case of `djpeg` bug-guided verification does not help much. This is because `djpeg` has a relatively smaller code base and contains fewer vulnerability labels, making bug-guided verification less utilized. These results are further evidence that *bug-guided verification can truly benefit fuzzing in terms of vulnerability finding thoroughness*.

C. Vulnerability Triage

The UBSan violations triggered by SAVIOR could lead to various consequences and some of them might be harmless. Therefore, we manually examine all the UBSan violations triggered by SAVIOR. These violations include those triggered in the 8 programs in Table V and also those from `mjs`, `catdoc`, and `c++filt`. We do not include the results of `mjs`, `catdoc`, and `c++filt` in the evaluation above, as all fuzzers trigger fewer than 10 UBSan violations. A small difference would result in a big variance in comparison.

Prog.	Number of reached UBSan labels					
	AFL	AFLGo	Angora	Driller	QSYM	SAVIOR
tcpdump	2029	1235	1333	1906	2509	2582
tiff2ps	748	927	770	931	852	970
readelf	91	79	102	104	106	183
xmllint	588	580	456	567	568	597
djpeg	2746	2588	2546	2713	2707	2746
tiff2pdf	1488	1467	919	1448	1369	1478
jasper	649	660	679	691	731	752
objdump	780	715	844	835	906	1039
Avg.	**1139**	**1031**	**956**	**1149**	**1218**	**1289**

TABLE VI: Number of unique UBSan labels reached by different fuzzers in 24 hours. On average SAVIOR reaches **19.68%** and **15.18%** more labels than Driller and QSYM.

Prog.	Improvements by bug-guided verification					
	AFL	AFLGo	Angora	Driller	QSYM	SAVIOR
tcpdump	+10/11%	+22/41.5%	+29/76.3%	+9/9.9%	+4/4%	+8/7%
tiff2ps	+4/133%	+0/0%	+3/42.9%	+0/0%	+0/0%	+0/0%
readelf	+10/82%	+9/72.2%	+16/107%	+9/68.4%	+8/63.2%	+7/29.2%
libxml	+4/33.3%	+4/33.3%	+5/166.7%	+4/33.3%	+4/33.3%	+0/0%
tiff2pdf	+5/50%	+1/7.7%	+4/44.4%	+3/27.2%	+5/62.5%	+0/0%
djpeg	+0/0%	+7/5.2%	+7/5.2%	+0/0%	+0/0%	+0/0%
objdump	+7/10.9%	+7/11.7%	+11/17.2%	+7/11.7%	+6/9.5%	+0/0%
jasper	+0/0%	+0/0%	+7/30.4%	+7/26.9%	+7/26.9%	+0/0%
Ave.	**+5/40.1%**	**+6/21.5%**	**+10/61.2%**	**+5/22.2%**	**+4.3/25%**	**+1.8/4.5%**

TABLE VII: New UBSan violations triggered with bug-guided verification in the evaluation with real-world programs. *"+X/Y%"* means *"X"* new violations are triggered, increasing the total number by *"Y%"*.

Program	Defect categories		Note	
	OOB	Logic Error	Exploitable*	Confirmed
tcpdump	6	102	6+	7
libjpeg	8	23	0+	N/A
objdump	41	4	4+	N/A
readelf	1	9	10+	3
libtiff	20	0	0+	N/A
jasper	21	2	2+	2
mjs	1	0	0+	1
catdoc	3	0	3+	1
c++filt	1	1	0	2
Total	**102**	**141**	**25+**	**16**

TABLE VIII: Triage of UBsan violations triggered by SAVIOR in 24 hours.

Triage Result: In total, we collect **481** UBSan violations and we manually classify them based on their consequences and present the results in Table VIII. Specifically, **102** of them lead to OOB reads/writes and **141** of them result in logic errors. Those logic errors consist of different categories, such as incorrect computation, wrong outputs, and polluted conditional variables. Among the **243** OOB and logic errors, **16** of them have been confirmed by the developers. Our further analysis so far reveals at least **25** of them are exploitable for goals such as information leak and control flow manipulation.

The remaining **238** cases are likely harmless according to our triage result. They mainly consist of the following categories: (1) the variables triggering UBSan violations are used as storage (*e.g.,* `int` as `char[4]`) instead of computation-related objects; (2) the affected variables expire immediately after the violations; (3) the program already considers the case of UBSan violations and has handlers.

Case Studies: From each of the three categories (OOB, logic errors, and those without harm), we pick a case and explain the details here. All the cases have been fixed.

The first case is an OOB in `readelf`. The code is shown below. The variable `inote.namesz` is copied from input. By making it equal to 0, (`inote.namesz` − 1) under-flows to the maximal unsigned value. It causes an OOB access to `inote.namedata`.

```
1 static bool process_notes_at(...){
2  //readelf.c:18303
3  if(inote.namedata[inote.namesz-1] != '\0')
4    ...
5 }
```

The second case is a logic error in `libtiff`. Variable `twobitdeltas[delta]` is controlled by user. With a specially crafted input, one can cause an overflow in the result of `lastpixel + twobitdeltas[delta]`, making `SETPIXEL` set the wrong pixel value to the decoded image.

```
1 static int ThunderDecode(...){
2  //tif_thunder.c:125
3  if((delta = ((n >> 4) & 3)) != DELTA2_SKIP)
4    SETPIXEL(op, lastpixel + twobitdeltas[
         delta]);
5   ...
6 }
```

The last case is harmless, as the program already considers overflow. This case locates in `libxml`. As shown below, with a special input, the variable `okey` can be overflowed. However, the program modulo `okey` with `dict->size` before using it, making the overflow harmless.

```
1 static int xmlDictGrow(...) {
2   // dict.c:417
3   okey = xmlDictComputeQKey(...);
4   key = okey % dict->size;
5   ...
6 }
```

VI. Related Works

The lines of works mostly related to our work include advanced fuzzing, concolic execution, the state-of-the-art hybrid testing techniques, and those that facilitate guided testing.

A. Advanced Fuzzing

Many recent works focus on improving the capability of code exploration in fuzzing. CollAFL [39] aims to reduce hash collision in coverage feedback to decrease false negatives. PTrix [30] enables path-sensitive fuzzing based on efficient hardware tracing. TFuzz [56] transforms tested programs to bypass complex conditions and improve code coverage, and later uses a validator to reproduce the inputs that work for the original program. To generate high-quality seeds, ProFuzzer [72] infers the structural information of the inputs. Along the line of seed generation, Angora [29] assumes a black-box function at each conditional statement and applies gradient descent to find satisfying input bytes. This method is later improved by NEUZZ [63] with a smooth surrogate

function to approximate the behavior of the tested program. Compared with these approaches, SAVIOR takes the bug-driven guidance to maximize bug coverage and verifies the (non-)existence of these bugs in the explored paths.

B. Concolic Execution

Symbolic execution, a systematic approach introduced in the 1970s [46, 48] for program testing, has attracted new attention due to the advances in satisfiability modulo theory [33, 34, 40]. However, classic symbolic execution has the problems of high computation cost and path explosion. To tackle these issues, Sen proposes concolic execution [59], which combines the constraint solving from symbolic execution and the fast execution of concrete testing. Concolic execution increases the coverage of random testing [41, 42] while also scales to large software. Hence, it has been adopted in various frameworks [26, 31, 60, 61]. Recently, concolic execution is also widely applied in automated vulnerability detection and exploitation, in which the concolic component provides critical inputs by incorporating security-related predicates [24, 28].

However, concolic execution operates based on emulation or heavy instrumentation, incurring tremendous execution overhead. Purely relying on concolic execution for code exploration is less practical for large software that involves large amounts of operations. In contrast, hybrid testing runs fuzzing for code exploration and invokes concolic execution only on hard-to-solve branches. This takes advantage of both fuzzer's efficiency and concolic executor's constraint solving.

C. Hybrid Testing

Majundar et al. [51] introduce the idea of hybrid concolic testing a decade ago. This idea offsets the deficiency of both random testing and concolic execution. Specifically, their approach interleaves random testing and concolic execution to deeply explore a wide program state space. Subsequent development reinforces hybrid testing by replacing random testing with guided fuzzing [55]. This approach could rapidly contributing more high-quality seeds to concolic execution.

Recently, DRILLER [66] engineers the state-of-the-art hybrid testing system. It more coherently combines fuzzing and concolic execution and can seamlessly test various software systems. Despite the advancement, DRILLER still achieves unsound vulnerability detection. DigFuzz [74] is a more recent work that tries to better coordinate the fuzzing and concolic execution components. Using a Monte Carlo algorithm, DigFuzz predicts the difficulty for a fuzzer to explore a path and prioritizes to explore seeds with a higher difficulty score.

Moreover, motivated by the growing demands in software testing, researchers have been reasoning the performance of hybrid testing. As commonly understood, hybrid testing is largely restricted by the slow concolic execution. To this end, QSYM [73] implements a concolic executor that tailors the heavy but unnecessary computations in symbolic interpretation and constraint solving. It leads to times of acceleration.

Differing from the above works that bring code-coverage improvement, SAVIOR changes the philosophy of hybrid testing. It drives the concolic executor on seeds with higher potential and guides the verification of the encountered vulnerabilities. This leads to quicker and better bug coverage.

D. Guided Software Testing

This line of research [25, 32, 43, 52] aims to guide the testing towards exploring specific code locations. Katch [52] prioritizes the seeds that approach patches to guide the symbolic executor. Together with three other guiding schemes, Katch can efficiently cover the target code. With a similar goal, AFLGO [25] calculates the distance from each code region to the targets (*e.g.,* vulnerable code regions or patches). In fuzz testing, AFLGO favors seeds that exercise code regions with smaller distances. Christakis et al. [32] proposes to prune paths in dynamic symbolic execution. It discards paths that carry properties that have been verified. However, the existing works generally prefer seeds that approach the targets quicker, which oftentimes carry shallow contexts. Instead, SAVIOR values all the seeds with high potential, creating various contexts to exercise the target code. This enables SAVIOR to outperforms these existing guided testing techniques in bug finding. Some other works use static analysis to label potential vulnerabilities, such as using data flow analysis to pinpoint data leaks [23], using slicing to mark use-after-free paths [38], and using taint analysis to mark possible races [50]. they then rely on subsequent symbolic execution to confirm detection. These analyses are complementary to SAVIOR. In addition, SAVIOR relies on fuzz testing to stably approach the to-be-verified paths, while others use heuristic based approaches to guide symbolic execution towards the marked label.

VII. CONCLUSION

We introduce SAVIOR, a new hybrid testing approach in this work. Unlike the mainstream hybrid testing tools which follow the coverage-driven design, SAVIOR moves towards being a bug-driven. We accordingly propose in SAVIOR two novel techniques, named *bug-driven prioritization* and *bug-guided verification*, respectively. On one hand, SAVIOR prioritizes the concolic execution to run seeds with more potentials of leading to vulnerabilities. On the other hand, SAVIOR examines all vulnerable candidates along the running program path in concolic execution. By modeling the unsafe conditions in SMT constraints, it solves for proofs of valid vulnerabilities or proves that the corresponding vulnerabilities do not exist. SAVIOR significantly outperforms the existing coverage-driven tools. On average, it detects vulnerabilities **43.4%** faster than DRILLER and **44.3%** faster than QSYM, resulting in the discovery of **88** and **76** more security violations in 24 hours.

ACKNOWLEDGMENTS

We would like to thank our shepherd Mathias Payer and the anonymous reviewers for their feedback. This project was supported by the Office of Naval Research (Grant#: N00014-17-1-2891, N00014-18-1-2043, and N00014-17-1-2787). Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding agency.

References

[1] "Aflgo source code," https://github.com/aflgo/aflgo.

[2] "American fuzzy lop," http://lcamtuf.coredump.cx/afl.

[3] "Angora source code," https://github.com/AngoraFuzzer/Angora.

[4] "angr/tracer: Utilities for generating dynamic traces," https://github.com/angr/tracer.

[5] "Apollo: an open autonomous driving platform," https://github.com/ApolloAuto/apollo.

[6] "Driller reading file patch," https://github.com/shellphish/driller/issues/48.

[7] "Driller stuck heuristic," https://github.com/shellphish/driller#the-stuck-heuristic.

[8] "Index of /gnu/binutils," https://ftp.gnu.org/gnu/binutils/.

[9] "jasper source code," https://github.com/mdadams/jasper/archive/master.zip.

[10] "Klee intrinsiccleaner," https://github.com/klee/klee/blob/master/lib/Module/IntrinsicCleaner.cpp.

[11] "libjpeg source code," https://www.ijg.org/files/jpegsrc.v9c.tar.gz.

[12] "Libtiff source code," https://download.osgeo.org/libtiff/.

[13] "libxml2 source code," http://xmlsoft.org/libxml2/libxml2-git-snapshot.tar.gz.

[14] "The myth of bug free software," https://www.betabreakers.com/the-myth-of-bug-free-software/.

[15] "Objdump overflow patch," https://sourceware.org/git/gitweb.cgi?p=binutils-gdb.git;a=commitdiff;h=f2023ce7.

[16] "Oss-fuzz - continuous fuzzing for open source software," https://github.com/google/oss-fuzz.

[17] "Qsym source code," https://github.com/sslab-gatech/qsym.

[18] "Run angora on lava dataset," https://github.com/AngoraFuzzer/Angora/blob/master/docs/lava.md.

[19] "T-fuzz source code," https://github.com/HexHive/T-Fuzz.

[20] "Tcpdump source code," http://www.tcpdump.org/release/.

[21] "Undefined behavior sanitizer - clang 9 documentation," http://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html#ubsan-checks.

[22] A. Afifi, D. Chan, J. J. Comuzzi, J. M. Hart, and A. Pizzarello, "Method and apparatus for analyzing computer code using weakest precondition," Feb. 22 2000, US Patent 6,029,002.

[23] S. Arzt, S. Rasthofer, R. Hahn, and E. Bodden, "Using targeted symbolic execution for reducing false-positives in dataflow analysis," in *Proceedings of the 4th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis, SOAP@PLDI 2015, Portland, OR, USA, June 15 - 17, 2015*, 2015, pp. 1–6.

[24] T. Avgerinos, S. K. Cha, B. L. T. Hao, and D. Brumley, "AEG: automatic exploit generation," in *Proceedings of the Network and Distributed System Security Symposium, NDSS 2011, San Diego, California, USA, 6th February - 9th February 2011*, 2011.

[25] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury, "Directed greybox fuzzing," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2017, pp. 2329–2344.

[26] J. Burnim and K. Sen, "Heuristics for scalable dynamic test generation," in *23rd IEEE/ACM International Conference on Automated Software Engineering, 15-19 September 2008, L'Aquila, Italy*, 2008, pp. 443–446.

[27] C. Cadar, D. Dunbar, and D. Engler, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*. USENIX Association, 2008, pp. 209–224.

[28] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley, "Unleashing mayhem on binary code," in *IEEE Symposium on Security and Privacy, SP 2012, 21-23 May 2012, San Francisco, California, USA*, 2012, pp. 380–394.

[29] P. Chen and H. Chen, "Angora: Efficient fuzzing by principled search," in *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2018, pp. 711–725.

[30] Y. Chen, D. Mu, J. Xu, Z. Sun, W. Shen, X. Xing, L. Lu, and B. Mao, "Ptrix: Efficient hardware-assisted fuzzing for cots binary," in *Proceedings of the 2019 ACM on Asia Conference on Computer and Communications Security*.

[31] V. Chipounov, V. Kuznetsov, and G. Candea, "S2E: A platform for in-vivo multi-path analysis of software systems," in *ACM SIGARCH Computer Architecture News*, vol. 39, no. 1. ACM, 2011, pp. 265–278.

[32] M. Christakis, P. Müller, and V. Wüstholz, "Guiding dynamic symbolic execution toward unverified program executions," in *Proceedings of the 38th International Conference on Software Engineering*. ACM, 2016, pp. 144–155.

[33] L. M. de Moura and N. Bjørner, "Satisfiability modulo theories: introduction and applications," *Commun. ACM*, vol. 54, no. 9, pp. 69–77, 2011.

[34] L. M. de Moura, B. Dutertre, and N. Shankar, "A tutorial on satisfiability modulo theories," ser. Lecture Notes in Computer Science, vol. 4590, 2007, pp. 20–36.

[35] W. Dietz, P. Li, J. Regehr, and V. Adve, "Understanding integer overflow in c/c++," in *Proceedings of the 34th International Conference on Software Engineering*, ser. ICSE'12, 2012, pp. 760–770.

[36] B. Dolan-Gavitt, P. Hulin, E. Kirda, T. Leek, A. Mambretti, W. Robertson, F. Ulrich, and R. Whelan, "Lava: Large-scale automated vulnerability addition," in *Proceedings of the 2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2016, pp. 110–121.

[37] H. Fan, F. Zhu, C. Liu, L. Zhang, L. Zhuang, D. Li, W. Zhu, J. Hu, H. Li, and Q. Kong, "Baidu apollo em motion planner," *arXiv preprint arXiv:1807.08048*, 2018.

[38] J. Feist, L. Mounier, S. Bardin, R. David, and M. Potet, "Finding the needle in the heap: combining static analysis and dynamic symbolic execution to trigger use-after-free," in *Proceedings of the 6th Workshop on*

Software Security, Protection, and Reverse Engineering, SSPREW@ACSAC 2016, Los Angeles, California, USA, December 5-6, 2016, 2016, pp. 2:1–2:12.

[39] S. Gan, C. Zhang, X. Qin, X. Tu, K. Li, Z. Pei, and Z. Chen, "Collafl: Path sensitive fuzzing," in *2018 IEEE Symposium on Security and Privacy (SP)*, vol. 00, pp. 660–677.

[40] V. Ganesh and D. L. Dill, "A decision procedure for bit-vectors and arrays," ser. Lecture Notes in Computer Science, vol. 4590, 2007, pp. 519–531.

[41] P. Godefroid, N. Klarlund, and K. Sen, "DART: directed automated random testing," in *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12-15, 2005*, 2005, pp. 213–223.

[42] P. Godefroid, M. Y. Levin, D. A. Molnar *et al.*, "Automated whitebox fuzz testing." in *NDSS*, vol. 8, 2008, pp. 151–166.

[43] S. Guo, M. Kusano, C. Wang, Z. Yang, and A. Gupta, "Assertion guided symbolic execution of multithreaded programs," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*, 2015, pp. 854–865.

[44] D. Harel, D. Kozen, and J. Tiuryn, "Dynamic logic," in *Handbook of philosophical logic*. Springer, 2001, pp. 99–217.

[45] C. Hathhorn, C. Ellison, and G. Rosu, "Defining the undefinedness of C," in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, 2015, pp. 336–345.

[46] W. E. Howden, "Symbolic testing and the DISSECT symbolic evaluation system," *IEEE Trans. Software Eng.*, vol. 3, no. 4, pp. 266–278, 1977.

[47] K. Jayaraman, D. Harvison, V. Ganesh, and A. Kiezun, "jfuzz: A concolic whitebox fuzzer for java," in *First NASA Formal Methods Symposium - NFM 2009, Moffett Field, California, USA, April 6-8, 2009.*, 2009, pp. 121–125.

[48] J. C. King, "Symbolic execution and program testing," *Commun. ACM*, vol. 19, no. 7, pp. 385–394, 1976.

[49] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, "Evaluating fuzz testing," *arXiv preprint arXiv:1808.09700*, 2018.

[50] P. Li, G. Li, and G. Gopalakrishnan, "Practical symbolic race checking of GPU programs," in *International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2014, New Orleans, LA, USA, November 16-21, 2014*, 2014, pp. 179–190.

[51] R. Majumdar and K. Sen, "Hybrid concolic testing," in *Software Engineering, 2007. ICSE 2007. 29th International Conference on*. IEEE, 2007, pp. 416–426.

[52] P. D. Marinescu and C. Cadar, "KATCH: high-coverage testing of software patches," in *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26, 2013*, 2013, pp. 235–245.

[53] P. E. McKnight and J. Najab, "Mann-whitney u test," *The Corsini encyclopedia of psychology*, pp. 1–1, 2010.

[54] D. Mu, A. Cuevas, L. Yang, H. Hu, X. Xing, B. Mao, and G. Wang, "Understanding the reproducibility of crowd-reported security vulnerabilities," in *Proceedings of the 27th USENIX Conference on Security Symposium*. USENIX Association, 2018, pp. 919–936.

[55] B. S. Pak, "Hybrid fuzz testing: Discovering software bugs via fuzzing and symbolic execution," *School of Computer Science Carnegie Mellon University*, 2012.

[56] H. Peng, Y. Shoshitaishvili, and M. Payer, "T-fuzz: fuzzing by program transformation," in *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2018, pp. 697–710.

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

[58] A. Rebert, S. K. Cha, T. Avgerinos, J. Foote, D. Warren, G. Grieco, and D. Brumley, "Optimizing seed selection for fuzzing," in *Proceedings of the 23rd USENIX Conference on Security Symposium*. USENIX Association, 2014, pp. 861–875.

[59] K. Sen, "Concolic testing," in *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*. ACM, 2007, pp. 571–572.

[60] K. Sen and G. Agha, "CUTE and jcute: Concolic unit testing and explicit path model-checking tools," in *Computer Aided Verification, 18th International Conference, Seattle, WA, USA, August 17-20, 2006, Proceedings*, 2006, pp. 419–423.

[61] K. Sen, D. Marinov, and G. Agha, "CUTE: a concolic unit testing engine for C," in *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2005, Lisbon, Portugal, September 5-9, 2005*, 2005, pp. 263–272.

[62] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "Addresssanitizer: A fast address sanity checker." in *USENIX Annual Technical Conference*. USENIX Association, 2012, pp. 309–318.

[63] D. She, K. Pei, D. Epstein, J. Yang, B. Ray, and S. Jana, "Neuzz: Efficient fuzzing with neural program smoothing," in *NEUZZ: Efficient Fuzzing with Neural Program Smoothing*. IEEE, 2018.

[64] R. M. Smullyan and R. Smullyan, *Gödel's incompleteness theorems*. Oxford University Press on Demand, 1992.

[65] V. C. Sreedhar, G. R. Gao, and Y.-f. Lee, "Incremental computation of dominator trees," in *ACM SIGPLAN Notices*, vol. 30, no. 3. ACM, 1995, pp. 1–12.

[66] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna,

"Driller: Augmenting fuzzing through selective symbolic execution." in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2016.

[67] Y. Sui and J. Xue, "Svf: interprocedural static value-flow analysis in llvm," in *Proceedings of the 25th International Conference on Compiler Construction*. ACM, 2016, pp. 265–266.

[68] A. Takanen, J. D. Demott, and C. Miller, *Fuzzing for software security testing and quality assurance*. Artech House, 2008.

[69] X. Wang, H. Chen, A. Cheung, Z. Jia, N. Zeldovich, and M. F. Kaashoek, "Undefined behavior: what happened to my code?" in *Asia-Pacific Workshop on Systems, APSys '12, Seoul, Republic of Korea, July 23-24, 2012*, 2012, p. 9.

[70] X. Wang, N. Zeldovich, M. F. Kaashoek, and A. Solar-Lezama, "Towards optimization-safe systems: analyzing the impact of undefined behavior," in *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*, 2013, pp. 260–275.

[71] Q. Yi, Z. Yang, S. Guo, C. Wang, J. Liu, and C. Zhao, "Eliminating path redundancy via postconditioned symbolic execution," *IEEE Trans. Software Eng.*, vol. 44, no. 1, pp. 25–43, 2018.

[72] 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 *ProFuzzer: On-the-fly Input Type Probing for Better Zero-Day Vulnerability Discovery*. IEEE.

[73] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim, "QSYM : A practical concolic execution engine tailored for hybrid fuzzing," in *Proceedings of the 27th USENIX Conference on Security Symposium*. USENIX Association, 2018, pp. 745–761.

[74] L. Zhao, Y. Duan, H. Yin, and J. Xuan, "Send hardest problems my way: Probabilistic path prioritization for hybrid fuzzing," in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2019.

APPENDIX A
SUPPLEMENTARY FIGURES AND EVALUATION DATA

A. Program Instrumentation

Figure 10 shows the UBSan-instrumented LLVM IR for the `objdump` defect in our motivating example, of which source code is presented in Figure 2. In Figure 10, we highlight the instrumentation with `!saviorBugNum` metadata for bug-driven prioritization.

B. UBSan Label Reduction

In the process of vulnerability labelling, SAVIOR also reduces labels that can be confirmed as false positives. Table IX shows the results of label reduction on our benchmark programs.

```
...
%23 = load
  ↪  %struct.dwarf_section*,%struct.dwarf_section**
  ↪  %section, align 8
%size10 = getelementptr %struct.dwarf_section,
  ↪  %struct.dwarf_section* %23, i32 0, i32 6
%24 = load i64, i64* %size10, align 8
  ↪  ; load value of section->size
%25 = call { i64, i1 }
  ↪  @llvm.uadd.with.overflow.i64(i64 %24, i64
  ↪  1)  ; section->size + 1
%26 = extractvalue { i64, i1 } %25, 0
 %27 = extractvalue  i64, i1  %25, 1
 %28 = xor i1 %27, true, !saviorBugNum !1
; check if the summation results in a carry
br i1 %28, label %cont, label %handler.add_overflow
cont:
  %call11 = call noalias i8* @malloc(i64 %26)
    ↪  #10  ; malloc(section->size + 1)
handler.add_overflow:   ; preds = %if.end6
  call void @__ubsan_handle_add_overflow()
  ...
```

Fig. 10: SAVIOR instrumentation of UBSan label.

Prog.	*Label reduction results*		
	Total UBSan Labels	Removed UBSan Labels	Percentage
tcpdump	13926	1924	13.8%
tiff2ps	1768	57	3.2%
readelf	2476	99	4.0%
xmllint	5258	195	3.7%
djpeg	9391	573	6.1%
tiff2pdf	3126	80	2.6%
jasper	3838	228	5.9%
objdump	9025	346	3.8%
Average	**6106**	**438**	**5.36%**

TABLE IX: Number of UBSan labels removed in our benchmark programs. On average, 5.36% of the labels are reduced.

C. LAVA-M Evaluation

In the evaluation with LAVA-M, bug-guided verification helps identify a group of LAVA bugs that are not listed. Table X shows the IDs of these LAVA bugs.

D. Real World Benchmark Evaluation

For a better reference of our evaluation with real-world programs, we summarize the number of triggered violations at the end of 24 hours in Table XII.

In addition, we also compare the UBSan violations triggered by SAVIOR and the other 5 fuzzers. The results are summarized in Table XI. In general, these fuzzers are exploring a similar group of UBSan violations. More importantly, for most of the cases, SAVIOR triggers a super-set of the violations that are made by the other fuzzers (in particular AFL and AFLGo). This indicates that SAVIOR has a better thoroughness in vulnerability finding.

APPENDIX B
TECHNICAL DISCUSSION AND FUTURE WORK

In this section, we discuss the limitations of our current design, insights we learned and possible future directions.

Over-approximation in Vulnerability Labeling: As explained in Section III, SAVIOR leverages sound algorithms to label vulnerabilities where the over-approximation may introduce many false-positive labels. This imprecision can

Program	*Bugs unlisted by LAVA-M but exposed by bug-guided verification*
base64	274, 521, 526, 527
uniq	227
md5sum	281, 287
who	1007, 1026, 1034, 1038, 1049, 1054, 1071, 1072, 117, 12 125, 1329, 1334, 1339, 1345, 1350, 1355, 1361, 1377, 1382 1388, 1393, 1397, 1403, 1408, 1415, 1420, 1429, 1436, 1445 1450, 1456, 1461, 16, 165, 169, 1718, 1727, 1728, 173 1735, 1736, 1737, 1738, 1747, 1748, 1755, 1756, 177, 181 185, 189, 1891, 1892, 1893, 1894, 1903, 1904, 1911, 1912 1921, 1925, 193, 1935, 1936, 1943, 1944, 1949, 1953, 197 1993, 1995, 1996, 2, 20, 2000, 2004, 2008, 2012, 2014 2019, 2023, 2027, 2031, 2034, 2035, 2039, 2043, 2047, 2051 2055, 2061, 2065, 2069, 2073, 2077, 2079, 2081, 2083, 210 214, 2147, 218, 2181, 2189, 2194, 2198, 2219, 222, 2221 2222, 2223, 2225, 2229, 2231, 2235, 2236, 2240, 2244, 2246 2247, 2249, 2253, 2255, 2258, 226, 2262, 2266, 2268, 2269 2271, 2275, 2282, 2286, 2291, 2295, 2302, 2304, 24, 2462 2463, 2464, 2465, 2466, 2467, 2468, 2469, 2499, 2500, 2507 2508, 2521, 2522, 2529, 2681, 2682, 2703, 2704, 2723, 2724 2742, 2796, 2804, 2806, 2814, 2818, 2823, 2827, 2834, 2838 2843, 2847, 2854, 2856, 2919, 2920, 2921, 2922, 294, 2974 2975, 298, 2982, 2983, 2994, 2995, 3002, 3003, 3013, 3021 303, 307, 3082, 3083, 3099, 312, 316, 3189, 3190, 3191 3192, 3198, 3202, 3209, 321, 3213, 3218, 3222, 3237, 3238 3239, 3242, 3245, 3247, 3249, 325, 3252, 3256, 3257, 3260 3264, 3265, 3267, 3269, 327, 334, 336, 338, 3389, 3439 346, 3466, 3468, 3469, 3470, 3471, 3487, 3488, 3495, 3496 350, 3509, 3510, 3517, 3518, 3523, 3527, 355, 359, 3939 4, 4024, 4025, 4026, 4027, 4222, 4223, 4224, 4225, 4287 4295, 450, 454, 459, 463, 468, 472, 477, 481, 483 488, 492, 497, 501, 504, 506, 512, 514, 522, 526 531, 535, 55, 57, 59, 6, 61, 63, 73, 77 8, 81, 85, 89, 974, 975, 994, 995, 996

TABLE X: IDs of unlisted bugs in LAVA-M that are triggered with bug-guided verification.

Prog.	*Difference of triggered UBSan violations*					
	AFL	AFLGo	Angora	Driller	QSYM	SAVIOR
tcpdump	+5/-43	+0/-61	+0/-76	+7/-30	+15/-28	+0/-0
tiff2ps	+0/-13	+0/-6	+0/-9	+0/-8	+0/-8	+0/-0
readelf	+0/-7	+1/-7	+4/-13	+2/-7	+2/-7	+0/-0
xmllint	+0/-6	+0/-6	+0/-15	+0/-6	+0/-6	+0/-6
djpeg	+0/-0	+0/-7	+0/-7	+0/-0	+0/-0	+0/-0
tiff2pdf	+0/-7	+0/-4	+5/-13	+0/-6	+0/-9	+0/-0
jasper	+2/-13	+0/-13	+1/-22	+0/-18	+0/-8	+0/-0
objdump	+14/-18	+10/-18	+16/-20	+10/-18	+12/-17	+0/-0

TABLE XI: Difference between violations triggered by SAVIOR and other fuzzers. (+X/-Y) means X violations are triggered by the fuzzer but not by SAVIOR and Y violations are triggered by SAVIOR but not by that fuzzer.

Prog.	*Number of triggered UBSan violations*					
	AFL	AFLGo	Angora	Driller	QSYM	SAVIOR
tcpdump	87	59	43	102	113	128
tiff2ps	3	10	7	8	8	16
readelf	14	16	14	15	16	22
xmllint	12	12	3	12	12	18
djpeg	141	134	134	141	141	141
tiff2pdf	13	13	9	13	10	17
jasper	33	31	23	26	26	44
objdump	64	60	64	60	63	79
Total	**367**	**335**	**297**	**377**	**389**	**465**

TABLE XII: Number of unique UBSan violations triggered by different fuzzers in 24 hours. In particular, **43.4%** and **44.3%** more violations than Driller and QSYM, respectively.

consequently weaken the performance of SAVIOR's prioritization. A straightforward reaction to this issue is to eliminate as many dummy labels as possible. In our design, we utilize a rule-based scheme to filter those false-positive labels in Section III-B. In the future, we plan to include more precise static analysis for finer-grained label pruning. For instance, the Stack system developed by Wang et. al [69, 70] and the approach proposed by Hathhorn et. al [45] can be incorporated into SAVIOR, which are complementary to UBSan in identifying code snippets that may lead to undefined behavior.

Prediction in Vulnerability Detection: Once reaching a potentially vulnerable program location in concolic execution, SAVIOR extracts the guarding predicates of the vulnerability label. However, these predicates may contradict the current path condition. In case of such contradiction, SAVIOR terminates the exploration of the labeling site immediately, since continuing the analysis cannot contribute to any valuable test input.

Moreover, in many cases, we can predict whether an execution path can trigger a vulnerability or not by studying the runtime information of previous executions. Also, more importantly, before that execution arrives the vulnerability site. To achieve this goal, we need a method to backwardly summarize path constraints from the labeled site to its predecessors in the explored paths. The core technique of this summary is the weakest precondition [44] (derived from the Hoare Logic) which has been applied to both sequential and concurrent program analysis domains [22, 43, 71].