Typestate-Guided Fuzzer for Discovering Use-after-Free Vulnerabilities

Haijun Wang
Ant Financial Services Group, China
& CSSE, Shenzhen University, China

Xiaofei Xie*
Nanyang Technological University
Singapore

Yi Li
Nanyang Technological University
Singapore

Cheng Wen
CSSE, Shenzhen University
Shenzhen, China

Yuekang Li
Nanyang Technological University
Singapore

Yang Liu
Nanyang Technological University
Singapore
Zhejiang Sci-Tech University, China

Shengchao Qin*
SCEDT, Teesside University, UK
CSSE, Shenzhen University, China

Hongxu Chen
Nanyang Technological University
Singapore

Yulei Sui
University of Technology Sydney
Australia

ABSTRACT

Existing coverage-based fuzzers usually use the individual control flow graph (CFG) edge coverage to guide the fuzzing process, which has shown great potential in finding vulnerabilities. However, CFG edge coverage is not effective in discovering vulnerabilities such as use-after-free (UaF). This is because, to trigger UaF vulnerabilities, one needs not only to cover individual edges, but also to traverse some (long) sequence of edges in a particular order, which is challenging for existing fuzzers. To this end, we propose to model UaF vulnerabilities as typestate properties, and develop a typestate-guided fuzzer, named UAFL, for discovering vulnerabilities violating typestate properties. Given a typestate property, we first perform a static typestate analysis to find operation sequences potentially violating the property. Our fuzzing process is then guided by the operation sequences in order to progressively generate test cases triggering property violations. In addition, we also employ an information flow analysis to improve the efficiency of the fuzzing process. We have performed a thorough evaluation of UAFL on 14 widely-used real-world programs. The experiment results show that UAFL substantially outperforms the state-of-the-art fuzzers, including AFL, AFLFast, FairFuzz, MOpt, Angora and QSYM, in terms of the time taken to discover vulnerabilities. We have discovered 10 previously unknown vulnerabilities, and received 5 new CVEs.

CCS CONCEPTS

• **Security and privacy → Software security engineering**.

*Corresponding authors: Shengchao Qin and Xiaofei Xie

ICSE '20, May 23–29, 2020, Seoul, Republic of Korea

ACM ISBN 978-1-4503-7121-6/20/05…$15.00
https://doi.org/10.1145/3377811.3380386

KEYWORDS

Fuzzing, Typestate-guided fuzzing, Use-after-Free vulnerabilities

ACM Reference Format:
Haijun Wang, Xiaofei Xie, Yi Li, Cheng Wen, Yuekang Li, Yang Liu, Shengchao Qin, Hongxu Chen, and Yulei Sui. 2020. Typestate-Guided Fuzzer for Discovering Use-after-Free Vulnerabilities. In *42nd International Conference on Software Engineering (ICSE '20), May 23–29, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3377811.3380386

1 INTRODUCTION

Software defects are pervasive [10] and may be exploited by malicious parties [38]. These software vulnerabilities have become fundamental threats against the software security. With the development of mitigation techniques [11, 33], many security vulnerabilities such as buffer overflow [12], have become harder to exploit. An exception is the use-after-free (UaF) vulnerability. There are still very few mitigation techniques deployed in production environments to defend against UaF vulnerabilities [42]. This has made UaF a significantly more vulnerable target for exploitation [19, 37, 44]. The UaF vulnerability happens when the memory is accessed after it has been freed previously, which may cause data corruption [38, 40], information leaks [19], denial-of-service [7], and arbitrary code execution attacks [5]. According to recent reports [19, 42], about 80% of the UaF vulnerabilities in the NVD database were rated critical or high in severity. In contrast, only about 50% of the heap buffer overflow vulnerabilities were considered as high severity.

Compared with other vulnerabilities (e.g., the stack/heap buffer overflow vulnerability), UaF vulnerabilities are generally considered more difficult to detect [19]. The main reason is that, to successfully trigger a UaF, a sequence of operations need to be executed in the specific order — first allocating the memory, then terminating the lifetime of the memory, and finally dereferencing the memory. These operations may not share locality in the code base, which makes the detection much more challenging.

Nevertheless, there exist some automated techniques for detecting and defending against UaF vulnerabilities. Static analysis-based techniques [30, 42] suffer from false positives, especially when dealing with real-world software systems. This is due to the fact that how to perform scalable and precise inter-procedural alias analysis

remains an open problem. Imprecise results from the alias analysis could also affect the effectiveness of some run-time detection mechanisms, such as FreeSentry [44] and DangNULL [19]. These techniques monitor the pointers to each memory location and replace them with invalid pointers once the memory is freed. If such an invalidated pointer is accessed, the program will subsequently crash, preventing an attacker from exploiting the vulnerability.

On the other hand, dynamic techniques, e.g., greybox fuzzing [18], which less likely produce false positives, have been shown effective in detecting memory-related vulnerabilities. Specially, AFL [47], libFuzzer [24] and the fuzzing infrastructure ClusterFuzz [17] have discovered more than 16,000 vulnerabilities in over 160 open source projects [17, 47]. However, existing techniques are not effective in detecting UaFs as shown in our experiments (c.f. Section 5.3). Existing coverage-based fuzzers, e.g., AFL [47], usually use the individual *Control Flow Graph (CFG) [36] edge coverage* to guide the fuzzing process. Unfortunately, CFG edge coverage is not so effective in discovering UaFs. This is because, to trigger UaF vulnerabilities, one needs not only to cover individual CFG edges, but also to traverse some long sequence of edges in a particular order, which is very difficult for existing coverage-based fuzzers. For example, the state-of-the-art grey-box fuzzers MOpt [25] and ProFuzzer [43] discover very few UaF vulnerabilities, according to their experimental results.

To address this challenge, we propose a typestate-guided fuzzer, named UAFL, for discovering vulnerabilities violating certain typestate properties. Our insight is that many common vulnerabilities can be seen as the violation of certain typesate properties. For example, the operation sequence, $\langle malloc \rightarrow free \rightarrow use \rangle$, is a witness of the typestate property violation shown in Fig. 3; therefore, it may trigger the UaF vulnerability. Similarly, $\langle nullify \rightarrow dereference \rangle$ is a witness for the null pointer dereferencing. Although our approach is generally applicable to all vulnerabilities which can be modelled by typestate property violations, we focus on one representative exemplar, UaF, in this paper. We first perform typestate analysis to identify *operation sequences* potentially violating the typestate properties. We then instrument the operation sequence coverage into the target program. Based on the information collected from the instrumentation, we propose two strategies to improve the effectiveness of the fuzzer: (1) we use the *operation sequence coverage* as the feedback to guide the test generation to progressively cover the operation sequences (c.f. Section 4.2), and (2) we deploy an *information flow analysis* to infer how test inputs affect the program states (c.f. Section 4.3), and use such information to design an efficient mutation strategy by avoiding unnecessary mutations.

We have implemented UAFL and evaluated it on 14 widely-used programs with diverse functionalities. UAFL significantly outperforms AFL [47], AFLFast [4], FairFuzz [21], MOpt [25], Angora [9], QSYM [46]. In particular, UAFL achieves, respectively, a speedup of 3.25×, 3.16×, 2.63×, 3.35×, 6.00×, and 3.80× to detect the vulnerabilities, compared to existing fuzzers AFL, AFLFast, FairFuzz, MOpt, Angora, and QSYM. Moreover, we have discovered 10 previously unknown UaF vulnerabilities in real-world programs, for them we have received 5 CVEs. All the new vulnerabilities have been confirmed.

The contribution of this work is summarized as follows.

```
1  void main() {
2    char buf[7];
3    read(0, buf, 7)
4    char* ptr1 = malloc(8);
5    char* ptr2 = malloc(8);
6    if(buf[5] == 'e')
7      ptr2 = ptr1;
8    if(buf[3] == 's')
9      if(buf[1] == 'u')
10       free(ptr1);
11   if(buf[4] == 'e')
12     if(buf[2] == 'r')
13       if(buf[0] == 'f')
14         ptr2[0] = 'm';
15   ...
16 }
```

Figure 1: An example simplified from UaF(CVE-2018-20623).

- We propose a typestate-guided fuzzer for discovering vulnerabilities violating certain typestate properties. It specifically addresses the challenge in detecting vulnerabilities triggered by specific sequences of operations (e.g., the UaF vulnerabilities), where we use operation sequence as guidance to progressively generate test cases violating the typestate properties.
- We design an effective mutation strategy customized for typestate-guided fuzzer and apply quantitative information flow analysis to help improve the overall performance of fuzzing process.
- We have implemented the proposed techniques as UAFL, and evaluated its effectiveness on a set of popular real-world programs. UAFL significantly outperforms the state-of-the-art fuzzers. We have discovered 10 previously unknown UaF vulnerabilities, 5 of which have been published as CVEs.

2 MOTIVATING EXAMPLE

In this section, we give an overview of UAFL with a motivating example, as shown in Fig. 1. This example is simplified from the real-world program *readelf*, which contains a UaF vulnerability (i.e., CVE-2018-20623). Fig. 2(a) shows its control flow graph (CFG), where the nodes represent the statements marked with their line numbers. The UaF vulnerability can be triggered when the statements (Lines 4, 7, 10 and 14) are executed temporally. Specifically, the pointer $ptr1$ points to the memory allocated at Line 4, and then $ptr1$ and $ptr2$ become aliases at Line 7. The memory pointed by $ptr1$ (and $ptr2$) is freed at Line 10, but accessed again at Line 14 by $ptr2$.

2.1 Existing Coverage-based Fuzzers

Existing coverage-based fuzzers usually utilize the CFG edge coverage to guide the fuzzing process, such as AFL [47]. In this section, we take AFL for example to illustrate how the CFG edge coverage is calculated in existing fuzzers. Given a program, AFL needs to identify the CFG edge (i.e., an edge from one basic block to another one). To this end, AFL first generates a random *ID* for each basic block of the program. Then, a CFG edge *ID* is calculated based on the *ID*s of two basic blocks. For example, regarding edge $A \rightarrow B$, its *ID* is calculated: $ID_{A \rightarrow B} = (ID_A >> 1) \oplus ID_B$, where the shift operation $>>$ is to preserve the directionality of the edge such that the *ID* of edge $A \rightarrow B$ is distinguishable from the *ID* of edge $B \rightarrow A$.

A shared memory *shared_mem* is used to count the hits of an edge. For example, $shared\_mem[ID_{A \rightarrow B}]$++ represents that hits of edge $A \rightarrow B$ are increased by 1. Actually, the hits of each edge is divided into 8 buckets: hit 1 time, 2 times, 3 times, 4-7 times, 8-15

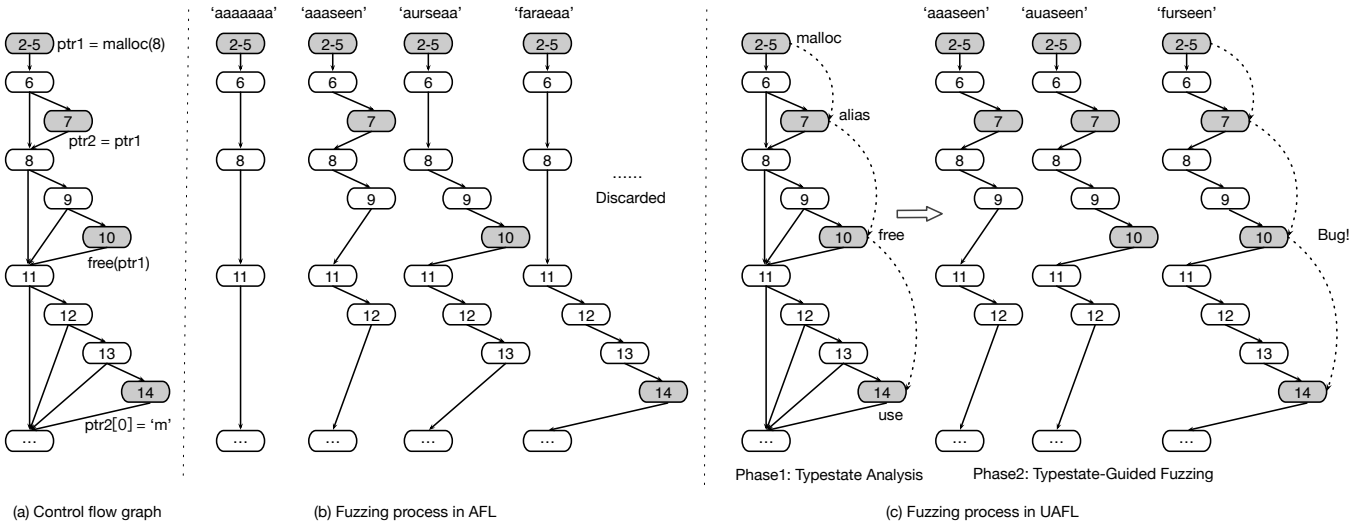

Figure 2: Fuzzing process of AFL and UAFL in the example of Fig. 1.

times, 16-31 times, 32-127 times, and 128-255 times. Given a test case, AFL generates new test cases by performing mutations on it. Basically, the new test case is considered as interesting and added into the test pool for further mutation, if it covers the edges not found by previous test cases or it touches the new buckets of edges. Otherwise, this test case is discarded.

2.2 Limitations of Coverage-based Fuzzers

Existing coverage-based fuzzers usually consider the CFG edge's coverage individually, and they are limited to track the coverage of a sequence of edges. For example, regarding a path $A \rightarrow B \rightarrow C$, AFL individually counts the hits of $A \rightarrow B$ and $B \rightarrow C$, but cannot track the hits of this path. Hence, existing coverage-based fuzzers are challenging to detect some vulnerabilities (e.g., UaF and double free), which violate the temporal memory safety *malloc* → *free* → *use*. This can be also specified as a typestate property, as shown in Fig. 3.

We take the program in Fig. 1 as an example, the UaF vulnerability can be triggered if Lines 4 → 7 → 10 → 14 are executed temporally (i.e., the grey nodes in Fig. 2(a)). Assume the initial seed is '*aaaaaaa*' and AFL generates three mutants '*aaaseen*', '*aurseaa*' and '*faraeaa*', as shown in Fig. 2(b). These four program paths cover all the CFG edges, but none of them can trigger the UaF vulnerability. Besides, since all CFG edges have been covered in these four test cases, their following mutants will be discarded as they cannot cover new CFG edges.

Considering the edge's coverage individually (e.g., AFL), it is highly difficult to generate a test case that can cover 4 → 7 → 10 → 14. We actually ran AFL on this example, and cannot find the UaF vulnerability within 24 hours.

2.3 Our Approach

Motivated by the aforementioned challenge, we propose a typestate-guided fuzzer, named UAFL, which aims to detect the vulnerabilities violating the typestate property. UAFL works in two phases: typestate analysis and typestate-guided fuzzing.

Phase1: Typestate Analysis. Based on the typestate property (e.g., UaF and double-free in Fig. 3), UAFL firstly performs the static typestate analysis to capture the operation sequences in the program, following the temporal relation violating the typestate property. For example, to discover the UaF vulnerability, UAFL identifies all operation sequences following the pattern *malloc* → *free* → *use*. As shown in typestate analysis step of Fig. 2(c), we present the identified operation sequence (i.e., 4 → 7 → 10 → 14), which satisfies the UaF pattern. This operation sequence first performs the memory allocation, then goes through the memory free, and finally reaches the memory use. It is worth noting that UAFL also performs the pointer alias analysis, e.g., pointers $ptr1$ and $ptr2$ may be aliases during the execution.

Phase2: Typestate-Guided Fuzzing. Guided by the operation sequences, UAFL generates test cases to progressively execute towards the operation sequences. UAFL first performs the instrumentation based on the identified operation sequences. The objective of instrumentation is to provide the feedback, which can guide UAFL to generate test cases that can execute towards the operation sequences. Fig. 2(c) shows the fuzzing process of UAFL. Assume that UAFL has also generated the test cases of Fig. 2(b) and covered all CFG edges. With the operation sequence (i.e., 4 → 7 → 10 → 14) as guidance, UAFL is able to generate new test cases that gradually cover the (entire) operation sequence. For example, based on test case '*aaaseen*', UAFL may generate a test case '*auaseen*'. This test case is discarded by AFL, since it does not cover new CFG edges compared to previous test cases '*aaaseen*' and '*aurseaa*'. However, as it covers edge 7 → 10 of operation sequence (i.e., *malloc* → *free*), UAFL adds it into the test pool for further mutation, By mutating test case '*auaseen*', UAFL may further generate new test case '$furseen$', which covers the operation sequence 4 → 7 → 10 → 14 (i.e., *malloc* → *free* → *use*). Thereby, the UaF vulnerability is discovered. We ran UAFL on this example, and discovered the UaF vulnerability within about 15 minutes.

3 TYPESTATE ANALYSIS

In this section, we elaborate on the typestate property and how to identify operation sequences potentially violating the UaF property with static typestate analysis.

3.1 Typestate Properties

To facilitate the illustration, we adopt a simplified programming language model defined as follows.

Definition 1 (Program [14]). *A program is a* Stmt *defined by the following context-free grammar,*

Stmt ::=	Var := Var	Var := *malloc*()		Var.op()
	Stmt; Stmt	*if* (∗) Stmt *else* Stmt		
	Label : Stmt	*goto Label*		

where ∗ denotes a non-deterministic branch, Var *are variables which reference to objects of type T,* Op *are operations supported by type T.*

We adopt the intuitive notion of a path p through a program P (or P-path): a valid sequence of statements starting at P's entry. A program path may contain operations on multiple objects, and we can extract multiple operation sequences from it, each per object. The *operation sequence* for objects is formally defined as follows.

Definition 2 (Operation Sequences (OSs) for Objects [14]). *Given a P-path p, $\mathcal{U}(p)$ denotes the set of object instances created during this execution, and for any object $o \in \mathcal{U}(p)$, $p[o]$ denotes the sequence of operations performed on o during execution of p.*

For example, assume that a P-path is $p = \langle a.malloc(), a.insert(), b.malloc(), b.free(), a.free()\rangle$, then we have $\mathcal{U}(p) = \{a, b\}$ and $p[a] = \langle a.malloc(), a.insert(), a.free()\rangle$. An operation sequence for a particular object can be checked against a typestate property, which is formally defined as follows.

Definition 3 (Typestate Property [15]). *A typestate property $\mathcal{P}$ is a finite state automaton $\mathcal{P} = (\Sigma, Q, \delta, init, Q \setminus \{err\})$, where Σ is the alphabet of observable operations, Q is the set of states, δ is the transition function mapping a state and an operation to a successor state, $init \in Q$ is a distinguished initial state, $err \in Q$ is a distinguished error state such that for every $\sigma \in \Sigma, \delta(err, \sigma) = err$, and all states in $Q \setminus \{err\}$ are accepting states.*

We say that q' is the successor of a state q on operation op when $\delta(q, \mathsf{op}) = q'$. Given a sequence of operations $\alpha = \langle \mathsf{op}_0, \ldots, \mathsf{op}_n\rangle$, we write $\alpha \in \mathcal{P}$ when α is accepted by $\mathcal{P}$, and we write $\alpha \notin \mathcal{P}$ when α is not accepted by $\mathcal{P}$.

Typestate Analysis. Regarding typestate properties, a set of observable operations accepted by the automaton, are valid traces. On the other hand, a set of operation sequences not accepted by the automaton, represent invalid traces which potentially violate typestate properties. Intuitively, vulnerabilities triggered by a sequence of operations can be captured by typestate properties. Given a typestate property $\mathcal{P}$, the *typestate analysis problem* for P is to determine whether there exists a path p such that $\exists o \in \mathcal{U}(p) : p[o] \notin \mathcal{P}$.

3.2 Use-After-Free Detection Problem

In this paper, we focus on detecting the UaF vulnerability, and Fig. 3 illustrates its typestate property. The alphabet for this problem consists of three operations, $\Sigma = \{malloc, free, use\}$. When the memory is allocated, it is in the *live* state, and can be accessed afterwards. Once the memory is freed, it becomes *dead*. Any subsequent use/free operation on this memory results in an *error* state. Notice that, the double-free vulnerability is a special case of UaF. *The UaF detection is to discover test cases that perform the operation sequences violating typestate property in Fig. 3.*

Algorithm 1: *Typestate Analysis for Use-After-Free*

input : A program P
output: A set of operation sequences S

1 $S \leftarrow \emptyset$;
2 $(S_M, M) \leftarrow find\_malloc(P)$;
3 **foreach** $(s_m, m) \in (S_M, M)$ **do**
4 $\quad A \leftarrow cal\_alias(m)$;
5 $\quad S_F \leftarrow find\_free(A, P)$;
6 $\quad S_U \leftarrow find\_use(A, P)$;
7 $\quad S \leftarrow S \cup \{\langle s_m, s_f, s_u\rangle | s_f \in S_F \wedge s_u \in S_U \wedge is\_reachable(s_m, s_f) \wedge is\_reachable(s_f, s_u)\}$;
8 **return** S;

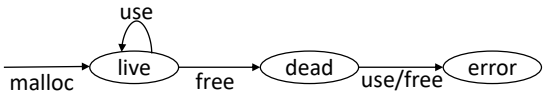

Figure 3: Typestate for use-after-free and double free.

Refer to the example in Fig. 1, a P-path $\langle 2-5, 6, 7, 8, 9, 10, 11, 12, 13, 14\rangle$ contains the operation sequence for the memory associated with $ptr1$: $\langle 4, 7, 10, 14\rangle$, which violates the given typestate property.

3.3 Use-after-Free Typestate Analysis

Given the typestate property of UaF vulnerability, UAFL performs the static typestate analysis to identify the operation sequences violating this property. Algorithm 1 shows the basic idea, which takes as the input a program P and outputs a set of operation sequences S violating the property.

For the program P, UAFL first finds all memory allocation statements S_M (Line 2), where M is a set of memory objects allocated at S_M. For each memory object m allocated at s_m (Line 3), UAFL calculates all the aliased pointers A that all point to the memory object m, with the pointer alias analysis [30, 32] (Line 4). Then, UAFL identifies all memory *free* statements that free the memory object m by an aliased pointer in A (Line 5). Following the same way, UAFL finds all memory *use* statements (Line 6). Finally, the operation sequence $\langle s_m, s_f, s_u\rangle$ is added into output S (Line 7), where s_m allocates the memory, s_f frees the memory which is reachable from s_m, and s_u uses the memory which is reachable from s_f.

Notice that, UAFL adopts the path-insensitive reachability analysis to perform typestate analysis, and may produce false positives. Thus, some operation sequences identified in S may not violate the typestate property. Furthermore, we employ the fuzzing technique that will confirm whether they actually violate the property.

Example 1. *In Fig. 1, two memory objects pointed by pointers $ptr1$ and $ptr2$ are allocated at Lines* 4 *and* 5. *Regarding $ptr1$, we identify the aliased pointer $ptr2$ at Line* 9. *Then, the memory is freed at Line* 10 *by $ptr1$. Lastly, Line* 14 *accesses the memory again by the aliased pointer $ptr2$. Since Line* 10 *is reachable from Line* 4 *and Line* 14 *is reachable*

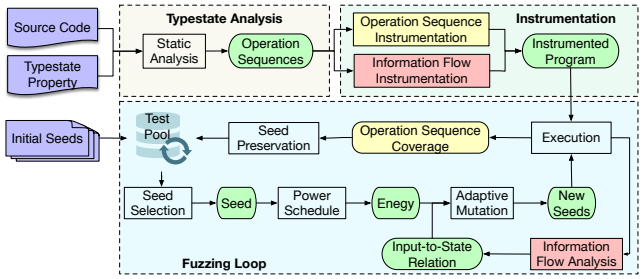

Figure 4: Workflow of typestate-guided fuzzing.

from Line 12*, we identify an operation sequence* $4 \rightarrow 10 \rightarrow 14$*. To be more precise, if the memory free and use statements use the aliased pointers, we incorporate the alias statement into the sequence. As a result, we obtain the sequence* $4 \rightarrow 7 \rightarrow 10 \rightarrow 14$ *(c.f. Fig. 2(a)), which violates the property of UaF.*

4 TYPESTATE-GUIDED FUZZING

Given the operation sequences (OSs) identified by the static typestate analysis, we perform OS-guided instrumentation to steer the fuzzing process, which can generate test cases to progressively cover the type-state statement sequences at runtime. UAFL mainly proposes two strategies to improve the effectiveness of the fuzzer.

(1) We propose the OS-guided mechanism, which can guide UAFL to generate test cases progressively covering the OSs. For example, to cover the sequence $4 \rightarrow 7 \rightarrow 10 \rightarrow 14$ in Fig. 2, UAFL gradually generates the test cases to cover $4 \rightarrow 7$, $4 \rightarrow 7 \rightarrow 10$, and $4 \rightarrow 7 \rightarrow 10 \rightarrow 14$ in turn.
(2) With the guidance developed, we still require an equipped mutation strategy to coordinate it. UAFL utilizes an information flow analysis to build the relationship between the input and variables in the program. Based on the relationship, UAFL adaptively mutates the input to change the states of variables. Take the program in Fig. 1 for example, the second and fourth bytes of input *buf* are relevant to Lines 8 and 9, and UAFL mutates them to generate a test case that can execute Line 10.

4.1 Fuzzing Workflow

Fig. 4 presents the fuzzing workflow. After identifying the OSs by typestate analysis, we instrument OS-guided information into the program. To achieve the two proposed strategies, we perform two kinds of instrumentation: operation sequence instrumentation and information flow instrumentation (c.f. Section 4.2). During the fuzzing loop, UAFL first selects a test case (i.e., the seed) from the test pool (c.f. Section 4.4). UAFL then measures the quality of this seed, and assigns an energy to it with the power scheduling strategy (c.f. Section 4.5). Next, we adopt the adaptive mutation strategy to mutate this seed and generate new seeds. The information flow analysis is used to produce an adaptive mutation strategy (c.f. Section 4.3). After the new seeds are generated, UAFL checks whether there is new OS coverage by running them. If so, the new seeds are considered interesting and added into the test pool for further mutation.

Algorithm 2: *UAFL Instrumentation*

input : a program P, and a set of operation sequences S
output: an instrumented program P'

1 let *OP_mem*, *OPE_mem* and *IFA_mem* be shared memory;
2 **foreach** $\langle s_0, \ldots, s_n \rangle \in S$ **do**
3 $\langle b_0, \ldots, b_m \rangle \leftarrow BB\_OP(\langle s_0, \ldots, s_n \rangle)$;
4 **foreach** $b_i \in \langle b_0, \ldots, b_m \rangle$ **do**
5 insert $OP\_mem[ID_{b_i}] \leftarrow 1$ into b_i;
6 **foreach** $b_i \in \langle b_1, \ldots, b_m \rangle$ **do**
7 let C_i be a set of conditional statements between b_{i-1} and b_i, and b_i is control-dependent on C_i;
8 **foreach** $c_j \in C_i$ **do**
9 let b_{c_j} be parent basic block of c_j;
10 $tID \leftarrow 0$;
11 **for** $0 \leq k < i$ **do**
12 **if** $OP\_mem[ID_{b_k}] == 1$ **then**
13 $tID \leftarrow tID \oplus ID_{b_k}$;
14 insert $OPE\_mem[tID \oplus ID_{b_{c_j}}]$ ++ into b_{c_j};
15 let c_j be comparison instruction *cmp a, b*;
16 insert $IFA\_mem[ID_{b_{c_j}}] \leftarrow a-b$ into b_{c_j};

4.2 Instrumentation

Algorithm 2 presents the details of our instrumentation. The inputs of the algorithm are a program P and a set of identified OSs S. The output is the instrumented program P'. From a high level, the instrumentation mainly includes three steps: 1) use the shared memory *OP_mem*[] to record whether the operation statements in the OSs are executed; 2) use the shared memory *OPE_mem*[] to record whether the OS edges are covered; and 3) use the shared memory *IFA_mem* to record the values of variables in the comparison instructions for the future information flow analysis (Line 1). It is worth noting, all elements of the three shared memory arrays are initialized as zero before each execution.

The instrumentation is conducted on the basic block, for each OS $\langle s_0, \ldots, s_n \rangle$ we first get its basic block sequence $\langle b_0, \ldots, b_m \rangle$ (Line 3). Note that the length of the basic block sequence may be less than the original OS, as some statements may belong to the same basic block. Regarding each b_i in the sequence, we instrument it with $OP\_mem[ID_{b_i}] \leftarrow 1$ (Line 5), representing whether b_i is executed or not. The ID_{b_i} is the *ID* of basic block b_i (c.f. Section 2.1).

In the second step, we perform the instrumentation to record the coverage of OSs. Notice that, the OS is different from the *P*-path (c.f. Section 3.1). Specifically, an OS edge may need to execute a sequence of CFG edges in *P*-path. For example, in Fig. 2(c), to cover the OS edge $7 \rightarrow 10$, it needs to execute the CFG edges $8 \rightarrow 9$ and $9 \rightarrow 10$. Therefore, it is highly difficult to cover the OS edge directly. Instead, we should first cover edge $8 \rightarrow 9$, and then $9 \rightarrow 10$.

To provide the fine-grained guidance for covering the OS edge, we consider its dominated basic blocks. For example, regarding $7 \rightarrow 10$ in Fig. 2(c), the parent basic blocks at Lines 8 and 9 in Fig. 1 are its dominated basic blocks. In particular, for each b_i in the sequence, UAFL analyzes the conditional statements C_i between b_{i-1} and b_i, b_i is control-dependent on C_i (Line 7). For each statement $c_j \in C_i$, UAFL gets its parent basic block b_{c_j} (Line 9), and instruments b_{c_j}

Algorithm 3: Mutation Probability Calculation

input : an program input $x[m]$, and variables $b_v[n]$ at n instructions
output: mutation probability $prob[n]$

1 **foreach** $i \in \{0, \ldots, m-1\}$ **do**
2 $X = \emptyset$, $Y = \emptyset$;
3 **foreach** $j \in \{0, \ldots, k\}$ **do**
4 $x' \leftarrow mutate(x[i])$;
5 $b'_v \leftarrow evaluate(x', b_v)$;
6 $X \leftarrow X \cup \{x'[i]\}$;
7 $Y \leftarrow Y \cup \{b'_v\}$;
8 $E(x[i]) \leftarrow \max_{j \in \{0,\ldots,n-1\}} IFStrength(x[i], b_v[j], X, Y[j])$;
9 $minE \leftarrow min(E(x[0]), \ldots E(x[m-1]))$;
10 $maxE \leftarrow max(E(x[0]), \ldots E(x[m-1]))$;
11 **foreach** $i \in \{0, \ldots, m-1\}$ **do**
12 $prob[i] \leftarrow \frac{E(x[i])-minE}{maxE-minE}$;

with $OPE\_mem[tID \oplus ID_{b_{c_j}}]$++ (Line 14). Specifically, the variable tID is used to represent the execution information of $\langle b_0, \ldots, b_{i-1} \rangle$ (Lines 10-13). If b_k is executed (i.e., $OP\_mem[ID_{b_k}] == 1$ at Line 12), its ID is embedded into tID (Line 13). As a result, tID can be used to guide the execution process toward b_{c_j}. tID would have a different value when a new basic block $b_k \in \langle b_0, \ldots, b_{i-1} \rangle$ is executed. Thus, the instrumentation can guide the fuzzing process to gradually cover the operation sequences, named *operation sequence feedback*.

In the third step, we preform the information flow instrumentation to conduct information flow analysis. We assume that the conditional statement is a comparison instruction (e.g., *cmp a, b*). UAFL records the values of $a-b$ by instrumenting $IFA\_mem[ID_{b_{c_j}}] \leftarrow a-b$ in the basic block b_{c_j} (Line 16).

Example 2. *Given the OS edge* $7 \rightarrow 10$ *in Fig. 2(c), we assume the basic block sequence is* $b_7 \rightarrow b_{10}$ *and their IDs are* ID_{b_7} *and* $ID_{b_{10}}$. *When* b_7 *is executed, we instrument* b_{10} *with* $OPE\_mem[ID_{b_7} \oplus ID_{b_{10}}]$++. *Otherwise, the instrumentation* $OPE\_mem[ID_{b_{10}}]$++ *is inserted into* b_{10}. *Thereby, our instrumentation can differentiate whether* b_7 *is executed or not once we reach* b_{10} *at runtime.*

4.3 Information Flow Analysis based Mutation

With the OSs and their dominated basic blocks identified, the next challenge is to find a way to perform the mutations so that the OSs can be covered quickly. For an instruction *cmp a, b*, the basic idea is that: we first identify which part of the input can change the values of $a - b$, then we can focus the resource on mutating this part of the input until its condition is satisfied. We adopt an information flow analysis to identify the relationship between the input and the program variables in the comparison instructions.

Information Flow Strength. We follow the information flow definition in [26], and compute the information flow strength from variable x to variable y as follows:

$$IFStrength(x, y, V_x, V_y) = H(x, V_x) - H(x|y, V_x, V_y) \quad (1)$$

where V_x and V_y are the value domains of variables x and y, respectively. $H(x, V_x)$ denotes the information entropy measuring the uncertainty of variable x.

$$H(x, V_x) = -\Sigma_{x_i \in V_x} P(x = x_i) log_2 P(x = x_i) \quad (2)$$

$H(x|y, V_x, V_y)$ denotes the conditional information entropy of variable x given the distribution of variable y:

$$H(x|y, V_x, V_y) = -\Sigma_{y_j \in V_y} P(y = y_j) * [\Sigma_{x_i \in V_x} P(x = x_i|y = y_j) log_2 P(x = x_i|y = y_j)] \quad (3)$$

Algorithm 3 mainly includes two steps. First, we compute the information flow strength between each byte of the input and variables (i.e., $a - b$) in the target comparison instructions (Lines 1-8). The higher the information flow strength, the stronger this byte influences the values of the variables. Then, we assign higher mutation possibility for these bytes of the input, as they are more likely to change the values of target instructions (Lines 9-12).

The inputs of the algorithm consist of the array $x[m]$ that is a m-bytes program input and the array $b_v[n]$ that includes variables in n comparison instructions, (i.e., $a - b$ in *cmp a, b*). Its output is an array $prob$ that contains the mutation probability for each byte of input x. For each byte of the input x (Line 1), we compute its information flow strength to each variable in b_v.

We use X and Y to store the sampling values for each byte of x and each variable of b_v (Line 2). We mutate each byte of x (i.e., $x[i]$) with k times (Line 3) and get k values for each variable of b_v. Specifically, after each mutation, we get a new mutant x' (Line 4). By running x' (Line 5), we can evaluate the values of variables b_v (c.f. $IFA\_mem[]$ at Line 16 in Algorithm 2). After generating k samples, we compute the information flow strength between $x[i]$ and each variable of b_v (Line 8). Notice that, for each $x[i]$, we compute n information flow strength values, and the maximum value is considered as the information flow strength of $x[i]$ (Line 8).

Based on the information flow strength, we calculate the mutation probability for each byte of input (i.e. $x[i]$) by the normalization (Line 9-12). Intuitively, if there is strong information flow between $x[i]$ and the program variables, it has a larger mutation probability.

Example 3. *In Fig. 1, given an input buf='aaaaaaa', assume the target instruction is* $buf[3] ==$ *'s' at Line* 8, *we consider the information flow strength between the bytes of input (i.e.,* $buf[3]$ *and* $buf[6]$*) and the variable (*$buf[3] -$ *'s') at Line* 8. *Assume each byte is mutated with 10 times (i.e.,* $k = 10$ *in Algorithm 3), then we figure out the information flow strength:* 3.3 *for* $buf[3]$ *and* 0 *for* $buf[6]$. *It suggests that there is a strong information flow between* $buf[3]$ *and the condition at Line* 8. *To cover this branch (i,e,* $buf[3] ==$ *'s'), we will assign more mutations on* $buf[3]$.

4.4 Seed Selection

The seed selection step is to select the next test case from the test pool such that it is more likely to cover the target OSs with some mutations on the selected test case. Motivated by the technique [8], UAFL provides a three-tiered queue, which classifies the generated seeds into different categories based on their scores. The seeds in the top-tiered queue have the highest priority, followed by the second-tiered, and finally the lowest-tiered.

These three tiers are designed based on the OS edge (e.g, $7 \rightarrow 10$ and $10 \rightarrow 14$ in Fig. 2) and CFG edge coverage (e.g., $8 \rightarrow 9$ and $9 \rightarrow 10$). Specifically, given a *newly generated* seed, (1) if it covers a new OS edge, we add it into the top-tiered queue, (2) if it covers a new CFG edge, we add it into the second-tiered queue, otherwise (3) the seed is added into the lowest-tiered queue. Intuitively, UAFL

favors the seeds that cover new OS edges, since it has more chances to cover the target OS. If the seed does not cover any new OS edge but covers a new CFG edge, which are control-dependent on the statements of the OS (refer to Line 7 in Algorithm 2), it still makes a small step towards covering the whole OS.

Example 4. *In Fig 2, a newly generated test case, which covers the new OS edge* $7 \rightarrow 10$*, is prioritized in the top-tiered queue since it is closer to cover the target OSs. However, if it does not cover* $7 \rightarrow 10$ *but CFG edge* $8 \rightarrow 9$*, it is also added into the second-tiered queue as it executes towards to* $7 \rightarrow 10$.

4.5 Power Scheduling

The power scheduling concerns about how many mutation chances are assigned to the given test case. A test case that has more chances to cover OSs with mutations should be assigned with more energy. Existing coverage-based fuzzers (e.g., AFL) usually calculate the energy for the selected test case as follows [23]:

$$energy(i) = allocate\_energy(q_i)$$

where i is the seed and q_i is the quality of the seed, depending on the execution time, branch edge coverage, creation time and so on.

UAFL considers the OS edge coverage and updates the existing power scheduling as follows:

$$energy(i)' = energy(i) * (1 + \frac{\#c\_OSe}{\#t\_OSe})$$

where $\#t\_OSe$ represents the total number of OS edges, and $\#c\_OSe$ denotes the number of OS edges that are covered by seed i. Intuitively, the number of covered OS edges implies how close the seeds are to OSs. Thus, UAFL assigns higher energy to seeds closer to OSs for them to have more chances to cover more OS edges with mutations.

5 EVALUATION

We have implemented the static typestate analysis on top of an interprocedural static value-flow analysis tool called SVF [32], which is able to perform scalable and precise interprocedural dependence analysis for C and C++ programs. This part takes about 4000 lines of C/C++ code. In the instrumentation component, we have designed three bulks of shared memory that tracks the execution information. The instrumentation consists of about 2000 lines of C/C++ code. The fuzzing engine is implemented based on the AFL version 2.51b, where we have implemented the information flow analysis based mutation. We added about 1000 lines of C/C++ codes into this part.

5.1 Evaluation Setup

In the experiments, we aim to answer the following questions:
RQ1 - What is the effectiveness of the static typestate analysis for UaF vulnerabilities?
RQ2 - How effective is UAFL in discovering UaF vulnerabilities?
RQ3 - How effective are the two strategies in UAFL, i.e., operation sequence feedback and information flow based mutation?
RQ4 - How is the code coverage relevant to UaF vulnerabilities?
Following Klees's suggestions in [18], we conduct a large-scale experiment and compare UAFL with state-of-the-art fuzzers.

5.1.1 Baseline Fuzzers. To evaluate the effectiveness of UAFL, we select 6 representative state-of-the-art fuzzers, which incorporate diverse advanced techniques to improve the effectiveness of fuzzing.

(1) **AFL** [47] is the most popular baseline fuzzer, which is studied in most coverage-based fuzzers.
(2) **AFLFast** [4] is an advanced variant of AFL with a better power scheduling that was incorporated into the later versions of AFL.
(3) **FairFuzz** [21] monitors the program executions, and computes the key patterns inside a seed to reach rare CFG edges.
(4) **MOpt** [25] adopts a customized Particle Swarm Optimization (PSO) algorithm to schedule the mutation operations, which enables more efficient discovery of vulnerabilities.
(5) **Angora** [9] utilizes taint analysis to track information flow, and then uses gradient descent to break through the hard branches.
(6) **QSYM** [46] is an effective symbolic execution assisted fuzzer.

5.1.2 Benchmark Programs. We consider the benchmark programs with the following factors: frequency of being tested in related work, popularity of being used by end users, and their functionality diversities. Finally, we select 14 widely-used programs as our benchmark, including the well-known development tools (e.g., *readelf 2.28*, *readelf 2.31*), code processing tools (e.g., *mjs*, *Mini XML*, *GNU cflow*, *nasm*), graphics processing libraries (e.g., *ImageMagick*, *jpegoptim*), compression tools (e.g., *lrzip* and *openh264*), data processing libraries (e.g., *libpff* and *liblouis*), an encryption key management tool (e.g., *boringssl*) and a Satisfiability Modulo Theories solver (e.g., *boolector*). For each program, we select the version that includes the UaF vulnerabilities (c.f. Column *Version* in Table 1).

5.1.3 Configuration Parameters. The effectiveness of fuzzers heavily replies on the random mutations, thus there may be performance deviation in the evaluation. Following Klees's suggestions in [18], we take three actions to mitigate performance deviation. First, we test each program for a longer time, until the fuzzer reaches a relatively stable state. In the paper, we run each fuzzer for 24 hours. Second, we perform each experiment for 8 times, and evaluate their statistical performance. Third, for the initial seeds, if the programs provide the sample seed inputs, we use them as the initial seeds. Otherwise, we randomly download some input files from the internet, according to the required input file formats. All the initial seeds can be found in our website [34]. All our experiments have been performed on machines with an Intel(R) Xeon(R) CPU E5-1650 v4 (3.60GHz) and 16GB of RAM under 64-bit Ubuntu LTS 18.04.

5.2 Static Typestate Analysis Statistics (RQ1)

In Table 1, we present the static typestate analysis results for benchmark programs. The first three columns denote the basic information of programs, including program names, their versions, and lines of code. We can see that the chosen programs have high diversities in terms of program size, from 2k to 1,781k lines of code.

Column *T_BB* lists the number of total basic blocks in the program. Column BB_{UAF} presents the number of basic blocks in OSs, which may violate the UaF typestate property. Specifically, UAFL instruments 19.2% of the basic blocks on average, to provide operation sequence feedback. Column BB_{IF} denotes the number of basic blocks in which we perform the information flow instrumentation. To calculate the information flow strength, UAFL performs

Table 1: Static Typestate Analysis Results

Program	Version	LoC	*T_BB*	BB_{UAF}	BB_{IF}	BB_{Free}	#OS	T(s)
readelf	2.28	1,781k	16,967	2,681 (15.8%)	1,103 (6.5%)	91	41,605	262
readelf	2.31	1,758k	19,973	3,647 (18.2%)	1,555 (7.8%)	98	130,102	508
jpegoptim	1.45	2k	634	36 (5.7%)	28 (4.4%)	5	44	1
liblouis	3.2.0	53k	2,957	486 (16.4%)	190 (6.4%)	8	422	18
lrzip	0.631	19k	9,356	1,051 (11.2%)	467 (5.0%)	6	313	150
Mini XML	2.12	15k	4,237	890 (21.0%)	788 (18.6%)	10	486	44
boringssl	—	162k	22,547	3,701 (16.4%)	3,265 (14.4%)	32	84,069	2,005
GNU cflow	1.6	50k	5,095	1,402 (27.5%)	751 (14.7%)	33	4330	30
Boolector	3.0.0	141k	26,866	11,511 (42.8%)	9,031 (33.6%)	4	28,586	2,387
openh264	1.8.0	143k	12,735	2,090 (16.4%)	927 (7.3%)	1	1,219	1,127
libpff	—	125k	18,569	6,371 (34.3%)	6,041 (32.5%)	60	20,865	122
mjs	1.20.1	40k	4,937	546 (11.0%)	343 (6.9%)	16	1,143	24
ImageMagick	7.0.8	485k	31,190	1,573 (5.0%)	1,336 (4.3%)	3	55,877	2,185
nasm	2.14	101k	13,965	3,812 (27.2%)	3,390 (24.2%)	2	3,357	2,210
Avg.	-	348k	13,573	2,842 (19.2%)	2,087 (13.3%)	26	26,601	1,148

Table 2: Time to Expose Uaf Vulnerabilities in 8 State-of-the-Art Fuzzers

Program	Vulnerabilities	Time Usage to Expose the Vulnerabilities (hours)							
		UAFL	$UAFL_{NoIF}$	AFL	AFLFast	FairFuzz	MOpt	Angora	QSYM
readelf-2.28	CVE-2017-6966	**0.59**	1.32	6.09	1.43	0.68	3.61	T/O	6.20
readelf-2.31	CVE-2018-20623	0.10	0.10	0.10	0.10	T/O	0.10	**0.02**	0.10
jpegoptim	CVE-2018-11416	**0.09**	0.10	0.59	0.88	1.08	1.49	T/O	1.95
liblouis	CVE-2017-13741	**1.11**	1.81	15.81	T/O	6.96	17.38	T/O	13.42
lrzip	CVE-2018-11496	**0.01**	0.01	0.01	0.01	0.01	0.01	0.01	0.01
Mini XML	CVE-2018-20592	**0.38**	0.93	1.28	2.59	0.54	16.7	T/O	18.99
boringssl	Google Test-suit	**0.33**	1.06	T/O	T/O	4.67	7.62	–	T/O
GNU cflow	uaf-issue-1	**1.80**	12.21	23.29	T/O	20.02	T/O	T/O	T/O
Boolector	uaf-issue-2	0.83	0.97	1.09	0.82	**0.39**	1.66	–	1.16
openh264	uaf-issue-3	**8.17**	13.00	15.80	11.15	8.17	15.39	T/O	18.45
libpff	uaf-issue-4	**1.39**	1.39	4.21	4.11	3.98	4.35	T/O	4.98
mjs	uaf-issue-5	**1.21**	1.23	3.10	3.02	1.45	4.6	T/O	6.71
ImageMagick	uaf-issue-6	**6.29**	13.92	T/O	T/O	T/O	T/O	T/O	T/O
nasm	CVE-2018-19216	**2.59**	4.69	8.32	3.45	2.86	11.46	2.75	9.64
	CVE-2018-20535	**17.03**	T/O	T/O	T/O	T/O	T/O	T/O	T/O
Missed Vulnerabilities		0	1	3	5	3	3	10	4
Avg. Time Usage		2.79 + 0.32	5.12 + 0.32	10.11	9.84	8.18	10.42	18.67	11.84
UAFL's Speedup		—	1.75×	3.25×	3.16×	2.63×	3.35×	6.00×	3.80×
$UAFL_{NoIF}$'s Speedup		—	—	1.86×	1.81×	1.50×	1.92×	3.43×	2.18×

* T/O means the fuzzer cannot discover vulnerabilities within 24 hours across 8 runs. When we calculate the average time usage, we replace T/O with 24 hours.
* Angora does not work on the programs *boringssl* and *Boolector*, denoted by '-', because it throws the exceptions during the instrumentation.

the instrumentation on an average of 13.3% of basic blocks. The experiment results show that, with less instrumentation (total 32.5% in both operation sequence and information flow instrumentation), UAFL can concentrate more power and energy on the OSs which may violate the typestate property of UaF.

Column BB_{Free} shows the number of static memory-free instructions. Column *#OS* presents the number of operation sequences identified by static typestate analysis. It reports on average 26, 601 OSs in each program. Since UAFL adopts the path-insensitive reachability analysis, it produces too many false positives. Thus, we employ the fuzzing to confirm whether these OSs really trigger UaF vulnerabilities. In the last column $T(s)$, we list the time overhead for static typestate analysis. In all programs, UAFL requires less than one hour, i.e., avg. 1, 148s (0.32 hour), to conduct the static typestate analysis. Compared to the long time fuzzing process (e.g., 24 hours), the time overhead for static typestate analysis is acceptable.

RQ1: The results show that the time overhead incurred in the static typestate analysis of UAFL is acceptable, with an average of 1,148s (0.32 hour). On average, 19.2% and 13.3% of basic blocks are instrumented, respectively, for operation sequence feedback and information flow analysis.

5.3 Vulnerability Detection Results (RQ2)

As suggested by Klees [18], the vulnerabilities found are ideal to measure the effectiveness of fuzzers. We evaluate the fuzzers with the time used for discovering UaF vulnerabilities.

Table 2 shows the time usage to discover the unique UaF vulnerability. The first two columns list the program names, and the vulnerability *ID*. The following columns presents the results for each fuzzer. Compared to state-of-the-arts fuzzers (e.g., AFL, AFLFast, FairFuzz, MOpt, Angora, and QSYM), UAFL takes much less time (i.e., avg. 2.79 hours) while others need about 10 hours. Even if the time overhead (i.e., avg. 0.32 hours) in static typestate analysis is added, UAFL is still more effective than others. Row *UAFL's Speedup* shows the average speedup achieved by UAFL: a speedup of 3.25×, 3.16×, 2.63×, 3.35×, 6.00×, and 3.80× respectively, compared with the other fuzzers.

For shallow vulnerabilities (e.g., *readelf-2.31*, *lrzip*), all fuzzers work well; but for deep/hard vulnerabilities, e.g., *liblouis*, *GNU cflow*, and *ImageMagick*, UAFL performs much better than the others. For example, UAFL takes about 1.80 hours to discover UaF in *GNU cflow*, while other fuzzers almost spend about 20 hours.

In addition, of the 15 vulnerabilities, UAFL performs the best in 13 (86.7%) vulnerabilities. In the program *readelf-2.31*, *Angora* performs the best, and *FairFuzz* achieves the best performance in *Boolector*. A close investigation reveals the reason behind is that these fuzzers' objectives coincidentally fit with the UaF. For example, *FairFuzz* aims to cover rare CFG edges and the memory-free statement in *Boolector* is under such rare CFG edges. Thus, *FairFuzz* has high chance to detect the UaF after covering the free statement. Different with other tools, UAFL is designed based on the typestate properties. Hence, UAFL performs more stably, and achieves better performance for most programs.

Row *Missed Vulnerabilities* shows the number of UaF vulnerabilities which cannot be discovered by each fuzzer. AFL, AFLFast, FairFuzz, MOpt, Angora and QSYM missed 3, 5, 3, 3, 10, and 4 vulnerabilities within 24 hours, respectively.

Statistical Test. To mitigate the randomness, we conducted the statistical test for the experiment results. The Vargha-Delaney statistic ($\hat{A}_{12}$) is a non-parametric measure of effect size, and usually used to measure the randomized algorithms [3]. Given the time usage in UAFL and other fuzzers, the $\hat{A}_{12}$ statistic measures the probability that UAFL performs better than others. Moreover, we also use Mann-Whitney U to measure the statistical significance of performance gain. When it is significant, we mark the $\hat{A}_{12}$ values in bold. In the column $\hat{A}_{12}$(UAFL) of Table 3, we can see that UAFL *significantly* performs better than other fuzzers in most cases.

RQ2: From Tables 2 and 3, UAFL performs significantly better than other fuzzers in most cases. UAFL achieves a speedup of 3.25×, 3.16×, 2.63×, 3.35×, 6.00×, and 3.80×, compared to AFL, AFLFast, FairFuzz, MOpt, Angora, and QSYM, respectively.

5.4 Evaluation of the Strategies (RQ3)

UAFL mainly adopts the operation sequence feedback and information flow analysis based mutation strategies to enhance the capability of UaF detection. To evaluate the effectiveness of each strategy, we configure a new fuzzer UAFL$_{NoIF}$, which only incorporates operation sequence feedback strategy but not the information flow analysis. The experiment results of UAFL$_{NoIF}$ is shown under the column UAFL$_{NoIF}$ in Table 2 and $\hat{A}_{12}$(UAFL$_{NoIF}$) in Table 3.

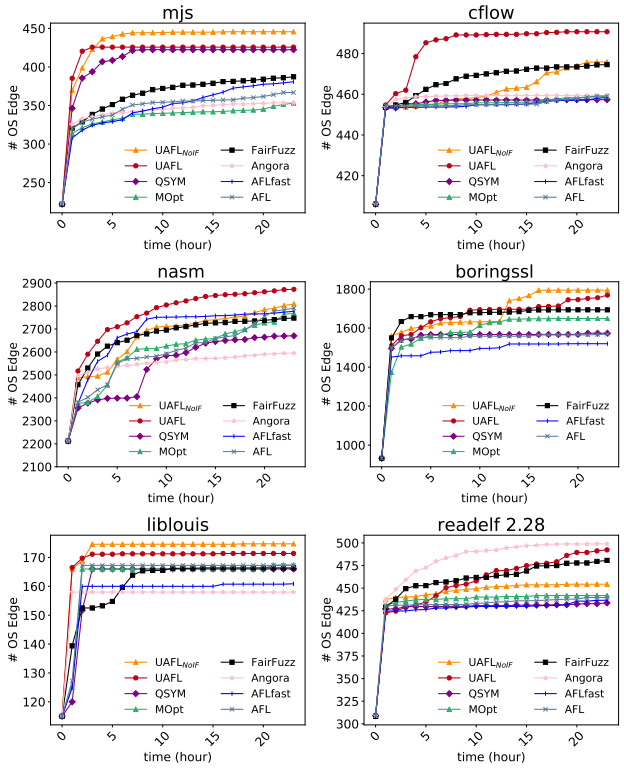

Figure 5: Code coverage relevant to operation sequences.

By comparing the results, it shows that UAFL$_{NoIF}$ is still better than other fuzzers in most cases. For example, UAFL$_{NoIF}$ averagely takes 5.12 hours to discover each vulnerability, achieving the speedups of 1.86×, 1.81×, 1.50×, 1.91×, 3.43×, and 2.18× over other fuzzers, respectively. Similarly, the statistical test also shows that UAFL$_{NoIF}$ is more effective in most cases. The results demonstrate the effectiveness of the operation sequence feedback.

By comparing the results of UAFL and UAFL$_{NoIF}$, we find that UAFL performs better than UAFL$_{NoIF}$ in each program. Specifically, UAFL achieves 1.75× improvement over UAFL$_{NoIF}$. It is worth noting, UAFL$_{NoIF}$ cannot find the UaF (i.e., CVE-2018-20535) in the program *nasm* within 24 hours. However, with the information flow based mutation incorporated, UAFL can discover the vulnerability with 17.03 hours. The results demonstrate the effectiveness of the information flow based mutation.

RQ3: The two strategies used in UAFL are both effective. The comparative results between UAFL$_{NoIF}$ and other tools show the effectiveness of operation sequence feedback. The comparative results between UAFL and UAFL$_{NoIF}$ show the contribution of the information flow analysis based mutation.

5.5 Code Coverage (RQ4)

Although there is no *fundamental* connection between maximizing code coverage and finding more vulnerabilities [18], the general efficacy of grey-box fuzzers over black-box ones means the certain correlation. It is believed that the code coverage makes sense as a secondary measure for the effectiveness of fuzzers. In this section,

Table 3: Statistical Test for Time-to-Exposure UaF Vulnerabilities

Program	Vulnerability	$\hat{A}_{12}$ (UAFL)						$\hat{A}_{12}$ (UAFL$_{NoIF}$)					
		AFL	AFLFast	FairFuzz	MOpt	Angora	QSYM	AFL	AFLFast	FairFuzz	MOpt	Angora	QSYM
readelf	CVE-2017-6966	**0.906**	**0.898**	**1.000**	0.609	**1.000**	**0.968**	**0.796**	0.546	**1.000**	0.453	**1.000**	**0.828**
readelf	CVE-2018-20623	0.500	0.500	0.500	0.500	**0.000**	0.500	0.500	0.500	0.500	0.500	**0.000**	0.500
jpegoptim	CVE-2018-11416	**0.995**	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**	**0.995**	**1.0**	**1.000**	**1.000**	**1.000**	**1.000**
liblouis	CVE-2017-13741	**0.828**	**0.937**	**0.851**	**1.000**	**1.000**	**0.984**	**0.875**	**0.937**	**0.867**	**1.000**	**1.000**	**0.968**
lrzip	CVE-2018-11496	0.500	0.500	0.500	0.500	0.500	0.500	0.500	0.500	0.500	0.500	0.500	0.500
Mini XML	CVE-2018-20592	**0.968**	**1.000**	**0.812**	**1.000**	**1.000**	**1.000**	0.617	**0.929**	**0.750**	**0.781**	**1.000**	**0.781**
boringssl	Google Test-suit	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**	**0.828**	**1.000**	**1.000**	**1.000**
GNU cflow	uaf-issue-1	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**	**0.937**	**0.968**	0.609	**0.968**	**1.000**	**1.000**
Boolector	uaf-issue-2	0.720	**1.000**	**0.030**	**0.880**	**1.000**	**0.780**	0.620	**1.000**	**0.020**	**0.820**	**1.000**	0.720
openh264	uaf-issue-3	**0.937**	**0.781**	**0.150**	**1.000**	**1.000**	**1.000**	0.687	0.359	**0.031**	0.640	**1.000**	**0.875**
libpff	uaf-issue-4	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**
mjs	uaf-issue-5	**0.980**	**0.980**	0.590	**1.000**	**1.000**	**1.000**	**0.880**	**0.890**	0.604	0.987	**1.000**	**0.987**
ImageMagick	uaf-issue-6	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**
nasm	CVE-2018-19216	0.960	0.600	0.560	**0.920**	0.600	0.800	**0.800**	0.319	0.280	**0.840**	0.280	0.800
	CVE-2018-20535	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**	0.500	0.500	0.500	0.500	0.500	0.500
Significant better ($\hat{A}_{12} > 0.71$, bold)		11/15	12/15	9/15	12/15	12/15	12/15	9/15	9/15	7/15	9/15	11/15	10/15

* We highlight the $\hat{A}_{12}$ in the bold if its corresponding *Mann-Whitney U* test is significant.

we only calculate the code coverage identified by the typestate analysis because 1) only they can help find the UaF vulnerabilities; and 2) the coverage can measure how close the generated test cases are to UaF vulnerabilities, thus can signify the quality of generated test cases. We evaluate the code coverage in UAFL, UAFL$_{NoIF}$, AFL, AFLFast, FairFuzz, MOpt, Angora, and QSYM. We only show here the results for 6 programs in which the UaF vulnerabilities are hard to detect, the rest can be found from our website [34].

Fig. 5 presents the code coverage achieved by 8 state-of-the-art fuzzers. In the early stage, all 8 fuzzers show the similar code coverage growth. As the time goes on, the code coverage in UAFL and UAFL$_{NoIF}$ grows faster than the other 6 fuzzers. At the end of 24 hours, UAFL and UAFL$_{NoIF}$ almost achieve the best code coverage among all eight fuzzers. The code coverage also reflects why UAFL and UAFL$_{NoIF}$ perform better in discovering vulnerabilities than others, as shown in Table 2. For other fuzzers, the results in AFL, AFLFast, MOpt, Angora, and QSYM are close to each other. FairFuzz averagely performs better than the other 5 fuzzers. After investigating the fuzzing process in FairFuzz, we find that some edges in OSs of UaF are rarely covered, which is in accordance with the target of FairFuzz. Although FairFuzz can cover some rare edges individually, it is still challenging to cover the whole operation sequences. Hence, it is less effective to discover UaF than UAFL. Another example is program *readelf-2.28*, where Angora achieves the best code coverage. However, Angora does not discover the UaF vulnerability as shown in Table 2. This is because Angora covers only individual edges but not the sequences.

RQ4: Compared to other fuzzers, UAFL and UAFL$_{NoIF}$ achieve better code coverage relevant to UaF vulnerability, which also explains why UAFL and UAFL$_{NoIF}$ are more effective in discovering UaF vulnerabilities.

5.6 Discussion and Threat to Validity

This paper evaluates our proposed typestate-guided fuzzer for detecting UaF vulnerabilities. Due to the high expressiveness of typestate property, our approach can be extended to detect other types of vulnerabilities. For example, we can extend UAFL to detect API-misuse [2], e.g., file read/write operations by customizing the typestate property. On the other hand, UAFL adopts the typestate analysis to identify operation sequences violating the typestate property, and then focuses its resource on these sequences, imposing less power for exercising other parts of the program. Thus, UAFL is complementary to existing coverage-based fuzzers to find the vulnerabilities that violate the typestate property.

One threat to validity is that, it may cause a certain sample bias when selecting programs to evaluate the capabilities of UAFL in discovering UaFs. To address this issue, we have selected a wide range of real-world programs, which have different functionalities and are frequently used in others' work. In addition, each experiment is conducted 8 times to mitigate the randomness factor. Due to the unique characteristics of UaF (i.e., malloc → free → use), we believe that the real-world UaF vulnerabilities found by UAFL are representative.

6 RELATED WORK

We discuss some closely related work in the following topic areas.

Typestate Analysis. Typestate analysis is a widely used technique in formal verification [13–15]. However, as proved by Field et al., the typestate verification problem becomes NP-hard for programs with maximum aliasing width of three and aliasing depth of two [14]. This limits the practical usefulness of typestate verification on large programs. To tackle this problem, Hua et al. introduced *machine-learning-guided typestate analysis* for static UaF detection [41]. They leverage machine learning techniques to reduce the overhead of typestate analysis, making it scalable to large-size programs. Chen et al. [45, 48] proposed a regular property guided symbolic execution to verify whether there is a path in the program satisfying the given property. These techniques are different from UAFL in two ways: (1) they employ symbolic execution while we enhance the fuzzing technique; and (2) they identify only a program path satisfying the property while we find all possible program paths violating the property. These works bring insights for UAFL, and we provide another approach to applying typestate analysis by incorporating it into fuzzing.

Use-After-Free Run-time Protection. Since UaF vulnerabilities are highly exploitable [40], many works are proposed to protect the program via early detection of UaF [5, 19, 35, 44]. Most of these works eliminate UaF vulnerabilities by capturing the existence of dangling pointers and disposing. This is because dangling pointers are the hotbed for UaF vulnerabilities. For example, in [19], a pointer

is nullified once the corresponding memory is freed. This effectively turns the UaF bug into a null-pointer dereference, which is much less harmful. These works are orthogonal to UAFL as they focus on capturing and monitoring the UaF behaviors while UAFL focuses on discovering and triggering UaF vulnerabilities.

Use-After-Free Static Detection. Besides the dynamic approach proposed in this paper, researchers also proposed several static approaches for UaF discovery [30, 41, 42]. These approaches rely heavily on pointer analysis, which bears a dilemma in nature. The dilemma is that we need to sacrifice a lot of scalability in exchange of a small amount of increment in precision and the result can still suffer from high false positives. Comparing with these static approaches, UAFL strikes a balance by having no false positive (every UaF found is associated with a proof-of-crash input) as well as good scalability (only light-weight static analysis is needed and the rest is handled by the dynamic part).

Greybox Fuzzing. Recently, lots of greybox fuzzing techniques have been proposed to detect various types of bugs [1, 4, 6, 9, 16, 21–23, 25, 28, 29, 31, 46]. These techniques either enhance the different components of the greybox fuzzer [4, 16, 21–23, 25] or combine greybox fuzzing with other techniques such as static analysis [23, 29], symbolic execution [31, 46], or taint analysis [9, 28]. These techniques are general purpose techniques which are not designed to detect a specific type of bugs with more effectiveness. With the experiments in Section 5, it is demonstrated that UAFL can substantially outperform these general purpose fuzzers in UaF detection. Besides the general purpose greybox fuzzers, there are also directed greybox fuzzers to quickly reach target locations [3, 8]. Comparing with the directed fuzzers, UAFL does not require prior knowledge about the locations of UaF bugs for reaching and detection. Other than the general purpose and directed greybox fuzzers, researchers also proposed fuzzing techniques for finding specific types of bugs, such as slowfuzz [27], perffuzz [20], MemLock [39]. However, none of them is designed for UaF bugs and to the best of our knowledge, UAFL is the first greybox fuzzer specialized in detecting UaF vulnerabilities.

7 CONCLUSION

In this paper, we propose a typestate-guided fuzzer UAFL for discovering vulnerabilities violating typestate properties, e.g. UaF vulnerabilities. We first employ a typestate analysis to extract operation sequences violating a given typestate property, which are then used to guide the fuzzing process to generate test cases progressively towards these operation sequences. We also adopt information flow analysis to guide the mutation process to improve the efficiency of the fuzzing process. Our experimental results have shown that UAFL substantially outperforms the state-of-the-art fuzzers, including AFL, AFLFast, FairFuzz, MOpt, Angora and QSYM, in discovering UaF vulnerabilities.

ACKNOWLEDGMENTS

This work was supported by Ant Financial Services Group through Ant Financial Research Program, the National Natural Science Foundation of China (Grant No. 61772347,61836005, 61972260), Singapore Ministry of Education Academic Research Fund Tier 1 (Award No. 2018-T1-002-069), the National Research Foundation, Prime Ministers Office, Singapore under its National Cybersecurity R&D Program (Award No. NRF2018NCR-NCR005-0001), the Singapore National Research Foundation under NCR Award Number NRF2018NCR-NSOE004-0001, and Australia Research Council (Grant No. DP200101328).

REFERENCES

[1] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. 2019. REDQUEEN: Fuzzing with Input-to-State Correspondence. In *26th Annual Network and Distributed System Security Symposium (NDSS'19)*. The Internet Society, San Diego, CA USA, 1–15.

[2] SungGyeong Bae, Hyunghun Cho, Inho Lim, and Sukyoung Ryu. 2014. SAFEWAPI: web API misuse detector for web applications. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 507–517.

[3] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed Greybox Fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM Press, 2329–2344.

[4] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2016. Coverage-based Greybox Fuzzing As Markov Chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM Press, 1032–1043.

[5] Juan Caballero, Gustavo Grieco, Mark Marron, and Antonio Nappa. 2012. Undangle: early detection of dangling pointers in use-after-free and double-free vulnerabilities. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*. ACM, 133–143.

[6] Hongxu Chen, Yuekang Li, Bihuan Chen, Yinxing Xue, and Yang Liu. 2018. FOT: A Versatile, Configurable, Extensible Fuzzing Framework *(FSE '18 tool demo)*.

[7] Haogang Chen, Yandong Mao, Xi Wang, Dong Zhou, Nickolai Zeldovich, and M Frans Kaashoek. 2011. Linux kernel vulnerabilities: State-of-the-art defenses and open problems. In *Proceedings of the Second Asia-Pacific Workshop on Systems*. ACM, 5.

[8] Hongxu Chen, Yinxing Xue, Yuekang Li, Bihuan Chen, Xiaofei Xie, Xiuheng Wu, and Yang Liu. 2018. Hawkeye: Towards a Desired Directed Grey-box Fuzzer. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM, New York, NY, USA, 2095–2108.

[9] Peng Chen and Hao Chen. 2018. Angora: Efficient fuzzing by principled search. In *2018 IEEE Symposium on Security and Privacy*. IEEE, 711–725.

[10] Yaohui Chen, Peng Li, Jun Xu, Shengjian Guo, Rundong Zhou, Yulong Zhang Zhang, Tao Wei, and Long Lu. 2020. SAVIOR: Towards Bug-Driven Hybrid Testing. In *2020 IEEE Symposium on Security and Privacy*. IEEE.

[11] Mauro Conti, Stephen Crane, Lucas Davi, Michael Franz, Per Larsen, Marco Negro, Christopher Liebchen, Mohaned Qunaibit, and Ahmad-Reza Sadeghi. 2015. Losing control: On the effectiveness of control-flow integrity under stack attacks. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 952–963.

[12] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. 1998. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks.. In *USENIX Security Symposium*, Vol. 98. San Antonio, TX, 63–78.

[13] Ankush Das, Shuvendu K. Lahiri, Akash Lal, and Yi Li. 2015. Angelic Verification: Precise Verification Modulo Unknowns. In *Proceedings of 27th International Conference on Computer Aided Verification*. San Francisco, CA, USA, 324–342.

[14] John Field, Deepak Goyal, G. Ramalingam, and Eran Yahav. 2003. Typestate Verification: Abstraction Techniques and Complexity Results. In *Static Analysis*, Radhia Cousot (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 439–462.

[15] Stephen Fink, Eran Yahav, Nurit Dor, G. Ramalingam, and Emmanuel Geay. 2006. Effective Typestate Verification in the Presence of Aliasing. In *Proceedings of the 2006 International Symposium on Software Testing and Analysis*. ACM, 133–144.

[16] Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuwen Tu, Kang Li, Zhongyu Pei, and Zuoning Chen. 2018. CollAFL: Path Sensitive Fuzzing. In *2018 IEEE Symposium on Security and Privacy*. IEEE Press, 1–12.

[17] Google Inc. 2019. *ClusterFuzz*. https://google.github.io/clusterfuzz/

[18] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating Fuzz Testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2123–2138.

[19] Byoungyoung Lee, Chengyu Song, Yeongjin Jang, Tielei Wang, Taesoo Kim, Long Lu, and Wenke Lee. 2015. Preventing Use-after-free with Dangling Pointers Nullification. In *22nd Annual Network and Distributed System Security Symposium*.

[20] Caroline Lemieux, Rohan Padhye, Koushik Sen, and Dawn Song. 2018. PerfFuzz: Automatically Generating Pathological Inputs. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 254–265.

[21] Caroline Lemieux and Koushik Sen. 2018. FairFuzz: A Targeted Mutation Strategy for Increasing Greybox Fuzz Testing Coverage. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ACM,

475–485.
[22] Yuekang Li, Bihuan Chen, Mahinthan Chandramohan, Shang-Wei Lin, Yang Liu, and Alwen Tiu. 2017. Steelix: Program-state Based Binary Fuzzing. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2017)*. ACM, New York, NY, USA, 627–637. https://doi.org/10.1145/3106237.3106295
[23] Yuekang Li, Yinxing Xue, Hongxu Chen, Xiuheng Wu, Cen Zhang, Xiaofei Xie, Haijun Wang, and Yang Liu. 2019. Cerebro: Context-aware Adaptive Fuzzing for Effective Vulnerability Detection. In *27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. Tallinn, Estonia.
[24] LLVM. 2015. *libFuzzer*. https://llvm.org/docs/LibFuzzer.html
[25] Chenyang Lyu, Shouling Ji, Chao Zhang, Yuwei Li, Wei-Han Lee, Yu Song, and Raheem Beyah. 2019. MOpt: Optimized Mutation Scheduling for Fuzzers. In *28th USENIX Security Symposium*. 1949–1966.
[26] Wes Masri and Andy Podgurski. 2009. Measuring the strength of information flows in programs. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 19, 2 (2009), 5.
[27] Theofilos Petsios, Jason Zhao, Angelos D. Keromytis, and Suman Jana. 2017. SlowFuzz: Automated Domain-Independent Detection of Algorithmic Complexity Vulnerabilities. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, New York, NY, USA, 2155–2168.
[28] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. 2017. VUzzer: Application-aware Evolutionary Fuzzing. In *NDSS*. https://www.vusec.net/download/?t=papers/vuzzer_ndss17.pdf
[29] Bhargava Shastry, Markus Leutner, Tobias Fiebig, Kashyap Thimmaraju, Fabian Yamaguchi, Konrad Rieck, Stefan Schmid, Jean-Pierre Seifert, and Anja Feldmann. 2017. Static program analysis as a fuzzing aid. In *International Symposium on Research in Attacks, Intrusions, and Defenses*. Springer, 26–47.
[30] Qingkai Shi, Xiao Xiao, Rongxin Wu, Jinguo Zhou, Gang Fan, and Charles Zhang. 2018. Pinpoint: Fast and precise sparse value flow analysis for million lines of code. In *ACM SIGPLAN Notices*, Vol. 53. ACM, 693–706.
[31] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Krügel, and Giovanni Vigna. 2016. Driller: Augmenting Fuzzing Through Selective Symbolic Execution. In *NDSS*.
[32] Yulei Sui and Jingling Xue. 2016. SVF: interprocedural static value-flow analysis in LLVM. In *Proceedings of the 25th international conference on compiler construction*. ACM, 265–266.
[33] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. 2013. Sok: Eternal war in memory. In *2013 IEEE Symposium on Security and Privacy*. IEEE, 48–62.
[34] UAFL. 2019. *Additoinal experiment results*. https://sites.google.com/view/uafl/
[35] Erik Van Der Kouwe, Vinod Nigade, and Cristiano Giuffrida. 2017. Dangsan: Scalable use-after-free detection. In *Proceedings of the Twelfth European Conference on Computer Systems*. ACM, 405–419.
[36] Haijun Wang, Yun Lin, Zijiang Yang, Jun Sun, Yang Liu, Jin Song Dong, Qinghua Zheng, and Ting Liu. 2019. Explaining Regressions via Alignment Slicing and Mending. *IEEE Transactions on Software Engineering* (2019), 1–1.
[37] Haijun Wang, Ting Liu, Xiaohong Guan, Chao Shen, Qinghua Zheng, and Zijiang Yang. 2016. Dependence guided symbolic execution. *IEEE Transactions on Software Engineering* 43, 3 (2016), 252–271.
[38] Haijun Wang, Xiaofei Xie, Shang-Wei Lin, Yun Lin, Yuekang Li, Shengchao Qin, Yang Liu, and Ting Liu. 2019. Locating vulnerabilities in binaries via memory layout recovering. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 718–728.
[39] Cheng Wen, Haijun Wang, Yuekang Li, Shengchao Qin, Yang Liu, Zhiwu Xu, Hongxu Chen, Xiaofei Xie, Geguang Pu, and Ting Liu. 2020. MemLock: Memory Usage Guided Fuzzing. In *2020 IEEE/ACM 42nd International Conference on Software Engineering*. Seoul, South Korea.
[40] Wen Xu, Juanru Li, Junliang Shu, Wenbo Yang, Tianyi Xie, Yuanyuan Zhang, and Dawu Gu. 2015. From collision to exploitation: Unleashing use-after-free vulnerabilities in linux kernel. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 414–425.
[41] Hua Yan, Yulei Sui, Shiping Chen, and Jingling Xue. 2017. Machine-Learning-Guided Typestate Analysis for Static Use-After-Free Detection. In *Proceedings of the 33rd Annual Computer Security Applications Conference*. 42–54.
[42] Hua Yan, Yulei Sui, Shiping Chen, and Jingling Xue. 2018. Spatio-temporal context reduction: A pointer-analysis-based static approach for detecting use-after-free vulnerabilities. In *2018 IEEE/ACM 40th International Conference on Software Engineering*. IEEE, 327–337.
[43] Wei You, Xueqiang Wang, Shiqing Ma, Jianjun Huang, Xiangyu Zhang, XiaoFeng Wang, and Bin Liang. 2019. Profuzzer: On-the-fly input type probing for better zero-day vulnerability discovery. In *2019 IEEE Symposium on Security and Privacy*. IEEE.
[44] Yves Younan. 2015. FreeSentry: protecting against use-after-free vulnerabilities due to dangling pointers. In *22nd Annual Network and Distributed System Security Symposiu*.
[45] H. Yu, Z. Chen, J. Wang, Z. Su, and W. Dong. 2018. Symbolic Verification of Regular Properties. In *2018 IEEE/ACM 40th International Conference on Software Engineering*. 871–881.
[46] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. 2018. QSYM: A practical concolic execution engine tailored for hybrid fuzzing. In *27th USENIX Security Symposium*. 745–761.
[47] Michal Zalewski. 2014. *American Fuzzy Lop*. http://lcamtuf.coredump.cx/afl/
[48] Yufeng Zhang, Zhenbang Chen, Ji Wang, Wei Dong, and Zhiming Liu. 2015. Regular property guided dynamic symbolic execution. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*. IEEE Press, 643–653.