MemLock: Memory Usage Guided Fuzzing

Cheng Wen
CSSE, Shenzhen University
Shenzhen, China

Haijun Wang
CSSE, Shenzhen University
Shenzhen, China

Yuekang Li
Nanyang Technological University
Singapore

Shengchao Qin
Teesside University,
Tees Valley, UK

Yang Liu
Nanyang Technological University
Singapore

Zhiwu Xu
CSSE, Shenzhen University
Shenzhen, China

Hongxu Chen, Xiaofei Xie
Nanyang Technological University
Singapore

Geguang Pu
East China Normal University
Shanghai, China

Ting Liu
Xi'an Jiaotong University
Xi'an, China

ABSTRACT

Uncontrolled memory consumption is a kind of critical software security weaknesses. It can also become a security-critical vulnerability when attackers can take control of the input to consume a large amount of memory and launch a Denial-of-Service attack. However, detecting such vulnerability is challenging, as the state-of-the-art fuzzing techniques focus on the code coverage but not memory consumption. To this end, we propose a memory usage guided fuzzing technique, named MemLock, to generate the excessive memory consumption inputs and trigger uncontrolled memory consumption bugs. The fuzzing process is guided with memory consumption information so that our approach is general and does not require any domain knowledge. We perform a thorough evaluation for MemLock on 14 widely-used real-world programs. Our experiment results show that MemLock substantially outperforms the state-of-the-art fuzzing techniques, including AFL, AFLfast, PerfFuzz, FairFuzz and QSYM, in discovering memory consumption bugs. During the experiments, we discovered many previously unknown memory consumption bugs and received 15 new CVEs.

1 INTRODUCTION

Time and space complexities are two main concerns in software design and development. If they are not implemented well, unexpected behaviors and even troublesome security issues can happen. In real-world programs, lots of such security vulnerabilities have been found (e.g., [14–20]). For example, if the termination conditions of recursive functions are not implemented correctly, infinite recursive function calls can occur and thus make the stack memory exhausted. The adversaries can exploit this vulnerability to launch a Denial-of-Service (DoS) attack with some well-crafted inputs [15, 18]. Recently, researchers have started to pay attention to these issues. For example, SlowFuzz [53], PerfFuzz [33] and ReScue [58] are developed to generate pathological inputs to stress the time complexity issues (i.e., algorithmic complexity vulnerabilities). However, it still leaves untouched for automatically generating pathological inputs to stress space complexity issues (namely memory consumption bugs) thus far.

ICSE 2020, 23-29 May 2020, Seoul, South Korea
2020. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Although a number of works (e.g., the popular fuzzing techniques [11, 25, 41, 56, 76]) have devoted to detecting memory issues, they mostly focus on memory corruption vulnerabilities such as buffer overflow and use-after-free. Memory corruption occurs in a program when the contents of the memory are modified due to some unexpected program behavior that exceeds the original intention of the program [60, 62]. When the corrupted memory contents are used later by the program, it may lead to unexpected behaviors (e.g., program crash). However, memory consumption bugs are essentially different from memory corruption vulnerabilities. As defined by CWE-400 [44], the software does not properly control the allocation and maintenance of a limited resource thereby enabling an actor to influence the amount of resources consumed, eventually leading to the exhaustion of available resources. To make it explicit, this paper focuses on three types of memory consumption bugs: *uncontrolled-recursion* [47], *uncontrolled-memory-allocation* [46], and *memory leak* [45]. Uncontrolled-recursion may exhaust stack memory when the program does not properly control the amount of recursion that takes place. Uncontrolled-memory-allocation refers to the situation whereby the program allocates memory based on an untrusted size value, but it does not validate or incorrectly validates the size, allowing arbitrary amounts of memory to be consumed. Moreover, if the software does not track and release allocated memory after it has been used, it causes a memory leak.

Existing detection techniques for memory consumption bugs usually use domain- or implementation-specific heuristics or rules [13, 21, 42, 65, 71]. For example, Radmin [21] learns and executes multiple probabilistic finite automata, and then confines the resource usage of target programs to the learned automata and detects resource usage anomalies at their early stages. Thus, their effectiveness heavily depends on the completeness of heuristics and rules. To create and maintain such rules requires substantial manual efforts and expertise. In this paper, we employ the grey-box fuzzing [76] technique to develop an automated and general technique to detect memory consumption bugs.

Grey-box fuzzing is one of the most effective techniques to find vulnerabilities [35, 37], which typically adopts the coverage information as guidance to explore different program paths. However, existing grey-box fuzzing techniques are not designed for detecting memory consumption bugs, because such bugs often depend not only on the program path but also on some interesting program

```
1  struct demangle_component *
2  cplus_demangle_type (struct d_info *di) {
3
4    // "peek" is a single character extracted from the input directly
5    char peek = d_peek_char (di);
6
7    switch (peek){
8      ...
9      case 'P':
10       ret = d_make_comp (di,
11         DEMANGLE_COMPONENT_POINTER,
12         cplus_demangle_type (di), NULL);
13       break;
14     case 'C':
15     ...
16   }
17   ...
18 }
```

Figure 1: Code Snippet from *cp-demangle.c* in *Binutils v2.31*

```
1  class EXIV2API DataBuf {
2  public:
3    // Constructor with an initial buffer size
4    explicit DataBuf(long size): pData(new byte[size]), size(size) {}
5    ...
6    byte* pData; // Pointer to the buffer
7    size_t size; // The current size of the buffer
8  };
9
10 void Jp2Image::readMetadata() {
11   while (io_->read((byte*)&subBox, sizeof(subBox)) ==
     ↪  sizeof(subBox) && subBox.length ) {
12     subBox.length = getLong((byte*)&subBox.length, bigEndian);
13     DataBuf data(subBox.length); // Allocation without checking
14     ...
15     io_->seek(position - sizeof(box) + box.length, BasicIo::beg);
16   }
17 }
```

Figure 2: Code Snippet from *jp2image.cpp* in *Exiv2 v0.26*

states in that path (i.e., amount of memory consumption). For example, the real-world program in Figure 2 allocates the memory at Line 4, however, this memory allocation may fail if no additional memory can be allocated for use. To detect this bug, the grey-box fuzzer needs to execute a program path that touches Line 4, as well as a large value for variable `size` to exceed the available heap memory. Existing coverage-based fuzzing techniques can easily cover Line 4, but it may be difficult to produce test cases that have a large value for variable `size`.

To address the aforementioned challenges, we present MemLock to enhance grey-box fuzzing to find memory consumption bugs. MemLock works in two steps. Firstly, MemLock performs the static analysis, which identifies the statements and operations relevant to memory consumption. We would qualitatively analyze the *call graph*, which determines the stack memory usage, and quantitatively analyze *memory usage operations*, which determines the heap memory usage. Besides, we also analyze the *control flow graph* of the program, which provides branch coverage for guiding to explore different program paths. With the memory consumption analyzed, MemLock then employs branch coverage as well as memory consumption information to guide the fuzzing process. The branch coverage information guides to explore different program paths, and the memory consumption information guides the program path to consume more and more memory. If an input covers new branch compared to previous inputs, it is considered as interesting and added into the seed queue. Besides, although an input has no new branch coverage, if it leads to more memory consumption, we also retain it as an interesting input through a novel seed updating scheme. This input can be further mutated so that the newly generated input leads to more memory consumption. After some mutations, MemLock is expected to generate an input whereby the memory consumption exceeds the available memory.

We have evaluated MemLock's effectiveness using a set of real-world open source programs. The experiment results show that MemLock substantially outperforms five state-of-the-art tools (i.e., AFL [76], AFLfast [8], PerfFuzz [33], FairFuzz [34] and QSYM [75]), in discovering the memory consumption vulnerabilities. In particular, MemLock finds 59.2% more unique crashes and 21.4% more vulnerabilities, than the second best counterpart. Besides, the generated test cases in MemLock usually lead to 150 times memory consumption compared to other five state-of-the-art tools. In addition, we have responsibly disclosed several previously unknown memory consumption bugs, and received 15 new CVE[1] for them, demonstrating MemLock's effectiveness in practice.

In summary, this paper makes the following contributions:

- We present MemLock, the first, to the best of our knowledge, dedicated fuzzing technique to automatically discover memory consumption bugs without requiring any domain knowledge.
- We design a new dimension of guidance engine to deeply exploit the memory consumption in a program path, which is complementary to the coverage guidance.
- We have implemented and evaluated MemLock on various widely-used real-world programs. The experimental results have shown that MemLock substantially outperforms five state-of-the-art fuzzing techniques in discovering memory consumption bugs.
- We have discovered 15 security-critical memory consumption vulnerabilities in widely-used real-world programs, and most of these vulnerabilities have been patched by the developers.

2 OVERVIEW

2.1 Motivating Examples

We first illustrate the limitations of existing coverage-based grey-box fuzzing techniques for detecting memory consumption bugs with two examples summarized from real-world vulnerabilities. We use the vulnerability CVE-2018-17985 [15] in Figure 1 to demonstrate a uncontrolled-recursion bug and CVE-2018-4868 [16] in Figure 2 to demonstrate an uncontrolled-memory-allocation bug.

In Figure 1, the function `cplus_demangle_type` recursively calls itself in line 12 when the input contains the character 'P'. The depth of recursion depends on the number of character 'P's in the input. With a sufficiently large recursive depth, the execution would run out of stack memory, causing stack overflow. To trigger a stack overflow, the fuzzer would need to generate inputs containing a large number of character 'P's.

However, existing coverage-based grey-box fuzzers do not have enough awareness about the change in recursive depth and solely use coverage information to retain interesting inputs. Take AFL as an example, it is aware of repeatedly executed CFG edges but only in a coarse manner. To be specific, AFL adopts the concept of "loop bucket" to retain interesting inputs (see Section 3.1). The loop bucket cannot tell the fine-grained change in recursive depth.

[1]The Common Vulnerabilities and Exposures (CVE) system provides a reference for tracking publicly known information-security vulnerabilities and exposures.

Specially, it does not differentiate the change when the recursive depth is greater than 255. Nevertheless, this number is still very far from causing stack exhaustion, which normally requires tens of thousands of recursive depth.

Therefore, to expose uncontrolled-recursion effectively, grey-box fuzzers need to have precise awareness about the stack memory consumption of the target program when executing an input.

Figure 2 demonstrates an uncontrolled-memory-allocation problem in *exiv2*. At line 11-12, when the program parses a `subBox` in `readMetadata()`, a length is extracted from the user inputs. Then the length is fed directly into `DataBuf()` at line 13. Finally, this value is used as the size of a memory allocation request at line 4. Note that the program does not check the size before allocating memory. By carefully handcrafting the input, an adversary can provide arbitrarily large values for `subBox.length`, leading to program crash (i.e., `std::bad_alloc`) or running out of memory. To trigger this problem, the fuzzer would need to generate inputs with a large `subBox.length`. For this purpose, the fuzzer needs to collect information about the value of `subBox.length` to retain the interesting inputs that can incur a large memory consumption.

However, existing coverage-based grey-box fuzzers lack awareness about the value of `subBox.length`. Therefore, they cannot effectively generate inputs causing `subBox.length` to become larger. Take AFL as an example, let us assume AFL now holds a seed input a which incurs the `subBox.length` of 100 and causes the function to enter the `while` at line 11 and eventually return at line 16. After some mutations, AFL may generate another input b which incurs the `subBox.length` of 10000 and also causes the function to enter the `while` at line 11 and return at line 16. We can clearly see that comparing with a, b consumes much more memory and is closer to running out of memory. However, AFL will discard input b and will not retain it as a seed because b does not bring new branch coverage. Consequently, AFL cannot detect this uncontrolled-memory-allocation problem effectively.

Therefore, to expose uncontrolled-memory-allocation effectively, grey-box fuzzers also need to have precise awareness about the amount of consumed heap memory of the target program when executing an input.

2.2 Approach Overview

Figure 3 shows the workflow of MemLock, which contains two main components: *static analysis* and *fuzzing loop*. In particular, the static analysis takes the *program source code* as the input, and generates three kinds of information (see Section 3.1): *control flow graph*, *call graph*, and *memory usage operations*. The static analysis in MemLock helps to decide *where to instrument* and *what to instrument*. The control flow graph information is used to collect the branch coverage; the call graph information aids to instrument the function call entries and returns. Based on the memory usage operation statements, MemLock instruments the locations of memory allocation and free operations.

Once the program is instrumented, MemLock enters the continuous fuzzing loop to detect memory consumption bugs (see Section 3.2). Given the initial seeds, MemLock selects a seed s from the seed pool. As for the seed s, MemLock generates the new inputs (test cases) using different mutation strategies. MemLock then

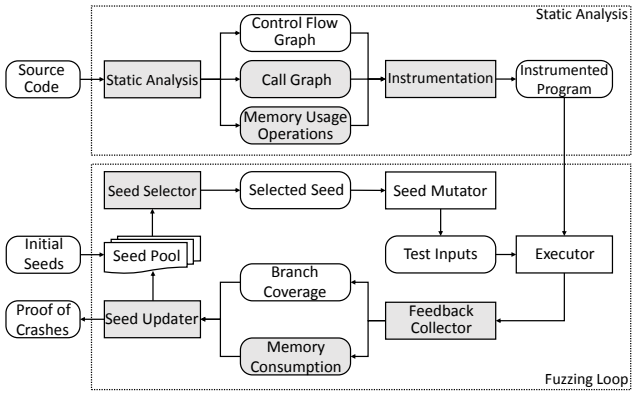

Figure 3: The overview of the proposed approach; grey rectangles denote the new features of MemLock.

runs the generated inputs against the instrumented program, and collects their memory consumption information (see Section 3.2.1) and branch coverage information. If the generated seeds consume more memory or have new branch coverage, they are retained as interesting seeds. MemLock adds them into the seed pool through a seed updating scheme (see Section 3.2.2). MemLock repeats this process until reaching time or resource budget limits.

Example in Figure 1. We illustrate MemLock using the example in Figure 1. Suppose the initial value of `peek` (obtained from function parameter *di* by function `d_peek_char` at Line 5) is 'a'. This value is general, unbiased for any special case. Through the coverage guidance, MemLock generates a new input i_1 that may produce the value 'P' for *peek* as it covers the different branch. When i_1 is further mutated, it generates i_2, which may produce four consecutive 'P's for *peek* (i.e., "PPPP") in its recursion. Since i_2 has different branch hits in the sense of "loop bucket" from i_1 , it is added into the seed pool. When i_2 is selected for mutation, it generates i_3 that may produce five consecutive 'P's for *peek* (i.e., "PPPPP") in its recursion. The coverage guidance uses the concept of "loop bucket", and considers that i_3 does not offer new branch coverage compared to i_1 and i_2. In this case, existing coverage-based grey-box fuzzers would discard i_3, and thus miss the chance to generate an input that can produce more consecutive 'P's. On the other hand, MemLock introduces memory consumption as the guidance, under which i_3 is considered to cause more memory consumption (than i_1 or i_2). Thus, it retains i_3 as an interesting test case, and adds it into the seed pool. It can further mutate i_3, and generate inputs that may produce more consecutive 'P's. After some mutations, MemLock may generate an input that would produce a sufficiently large number of consecutive 'P's (i.e., "PPP…") to run out the stack memory.

Example in Figure 2. For illustration, let us assume that the available heap memory is 10000 bytes. Suppose the initial value of `subBox.length` is 100, which is produced from user input at Lines 11-12. At Line 13 in Figure 2, the memory is allocated successfully, and the program executes the true branch of the `while` statement at Line 11. Based on the coverage guidance, MemLock performs the mutation and can generate a new input i_1 that produces a larger value for `subBox.length`. In this case, we assume the value is 150. The input i_1 still executes the true branch of the `while` statement, and thus there is no new branch coverage. At this time,

the coverage-based grey-box fuzzers would discard i_1, therefore missing the chance to generate an input consuming more memory. On the other hand, MemLock's memory consumption guidance considers that i_1 consumes more memory (i.e., 150 > 100), and keeps it as an interesting input. When i_1 is further mutated, MemLock can generate an input (e.g., $len = 250$) that consumes more memory. After some mutations, MemLock can generate an input (e.g., $len = 11000$) that runs out of memory.

Note that we haven't elaborated memory leaks separately as MemLock deals with them in the same way as uncontrolled-memory-allocation, using the same memory usage guidance during fuzzing.

3 METHODOLOGY

3.1 Static Analysis

The static analysis in MemLock decides how to instrument the target program. Based on the instrumentation, MemLock collects the guidance information, and then uses it to drive the fuzzing process. After analyzing the control flow graph, MemLock instruments the target program to capture branch (edge) coverage, guiding program path explorations. Additionally, based on the qualitative and quantitative analysis of *call graph* and *memory usage operations*, it also instruments the target program to collect the memory consumption information, guiding the fuzzing process towards consuming more memory for each program path.

3.1.1 Control Flow Graph. MemLock collects branch coverage information in the control flow graph (CFG) of the program to guide program path explorations as AFL [76]. It inserts instrumentation into every branch of the program CFG, assigning a pseudo-unique *ID* to every branch. During program execution, the instrumentation uses an 8-bit *counter* to keep track of the number of times that a branch has been executed. MemLock groups the hit counts of each branch execution into several buckets to denote different magnitudes[2]. Consequently, the branch coverage information in an executed program path can be defined as follows.

Definition 3.1 (Trace Bits [76]). *For an executed program path, its trace bits are represented by an 8-bit array with size 2^K, and the value of the ID^{th} element is stored in an 8-bit counter (In AFL, $K = 16$).*

The trace bits record the accumulated branches executed in a program path, and they can represent a program path roughly.

Definition 3.2 (Path-ID). *For an executed program path, its path-ID is the hash value of its trace bits (see Definition 3.1).*

3.1.2 Call Graph. In addition to branch coverage, MemLock also collects the memory consumption information. One important construct that may cause a large bulk of stack memory consumption is the recursive function call. When a function call occurs, the program automatically allocates the stack memory for use (e.g., local variables). On the other hand, when a function call is finished (returned), the program automatically reclaims the allocated stack memory for reuse. To monitor the stack memory consumption of function calls, MemLock injects the instrumentation into both the entry and the exit of the function call.

[2]In AFL, the hit counts of each branch execution are divided into 8 buckets: 1 time, 2 times, 3 times, 4-7 times, 8-15 times, 16-31 times, 32-127 times, and 128-255 times [70].

We use *ft* to denote the length (i.e., consumption) of call stack during the program execution. This value changes with the execution of the program. When the program execution enters a function, the value *ft* is increased by one; likewise, when a function call is returned, the value *ft* is decreased by one. In the following, we use *fm* to denote the peak value of *ft* during the program execution. The value *fm* thus qualitatively reflect the maximum (stack) memory consumption by recursive function calls during the program execution. We do not differentiate the memory consumption caused by different functions, because usually the stack memory can be exhausted only under infinite recursive function calls. Thus, we only need the peak length of call stack to guide MemLock to approach infinite recursive function calls.

3.1.3 Memory Usage Operations. Memory usage operation statements (e.g. *malloc* and *free*) may also contribute to the consumption of a large bulk of memory. In a program path, the memory operation statements may be affected by the program inputs. When this happens, it is possible to guide this program path to consume more memory by controlling the program inputs. To this end, MemLock uses instrumentation to quantitatively obtain the size of the memory operation. Due to the lack of freed memory size in deallocation statements, MemLock maps them to their corresponding allocation statements to obtain the size of the freed memory.

In particular, we insert instrumentation into the memory allocation/deallocation functions in the standard libraries, and obtain its parameters and return value. The reason is that the memory is allocated by some standard library functions [1, 42], e.g., *malloc*, *calloc*, *realloc*, and *new*. On the other hand, the program may also free the memory using the standard library function such as *free* and *delete*. Even when the program uses a user-customized memory usage operation function [29], it still relies on standard library functions to operate a larger bulk of memory. Thus, we do not need to consider the user-customized memory usage operations in practice.

We use *ot* to denote the amount of memory consumed by memory operations in a program path. When the program allocates ot' bytes memory, the value *ot* is increased by ot'; likewise, if it frees ot' bytes memory, the value *ot* is decreased by ot'. In the following, we use the *om* to represent the peak value of *ot* during the program execution. The value *om* evaluates the memory consumption in a program path by memory usage operation statements. By using *om* as the guidance, MemLock can mutate the program inputs and gradually increase the peak value of memory consumption in a program path.

3.2 Fuzzing Loop

Algorithm 1 shows the high-level procedures of MemLock. The intuition of the algorithm is that, for each input t in the seed pool, MemLock decides whether to mutate it based on a selection probability. If so, MemLock mutates t and generates a set of child inputs. Then, MemLock runs each child input and monitors their executions. If a child input has new coverage or consumes more memory (see Definitions 3.3 and 3.4), it is retained as an interesting input. While this process is similar to the process of traditional coverage-based grey-box fuzzers (e.g., AFL), the main difference is that MemLock additionally adopts memory consumption guidance to retain interesting inputs.

Algorithm 1: Memory Usage Guided Fuzzing

input : an instrumented program P, and set of initial seeds T
output: test cases S triggering memory consumption bugs

```
1  S ← Φ;
2  Queue ← T;
3  while time and resource budget do not expire do
4      for each input t in Queue do
5          if with probability a to select t then
6              numChildren ← AssignEnergy(t);
7              for 0 ≤ i < numChildren do
8                  child_i ← Mutate(t);
9                  (traceBits_i, fm_i, om_i) ← Run(child_i, P);
10                 k = Hash(traceBits_i);
11                 if it triggers memory consumption bugs then
12                     S ← S ∪ child_i;
13                 else
14                     if NewCov(traceBits_i) then
15                         Queue ← Queue ∪ child_i;
16                     if NewMax(fm_i, om_i) then
17                         Queue ←
                             Update(child_i, fmMap[k], omMap[k]);
18 return S
```

The algorithm takes the instrumented program P (see Section 3.1) and a set of initial seeds T as the inputs, and outputs a set of test cases S that trigger the memory consumption bugs. The variable *Queue* represents the seed pool, and is initialized as the initial seeds T at Line 2. MemLock first selects an input t from the seed pool *Queue* (Line 4), and computes its probability on whether or not to be mutated at Line 5 (see Section 3.2.1). Upon deciding to mutate the input t, MemLock assigns the energy (i.e., *numChildren*) to it at Line 6, which determines the number of children to produce from t. MemLock uses the same heuristics to determine *numChildren* as AFL [76]. It produces more children for inputs that have wider code coverage or that are discovered later in the fuzzing process. At Lines 4-17, MemLock mutates the input t to generate *numChildren* children, monitors their executions, and determines their affiliations. MemLock first performs mutation to generate the new input $child_i$ (Line 8). At Line 9, MemLock then runs the input $child_i$ on the instrumented program P, and collects its branch coverage (i.e., $traceBits_i$), function memory consumption (i.e., *fm*), and operation memory consumption (i.e., *om*), respectively.

If the input $child_i$ triggers memory consumption bugs (how to determine memory consumption bugs, see Section 4.1), it is added into the output S (Line 12). Otherwise, MemLock analyzes its branch coverage and memory consumption (Line 14 and 16). If it has new branch coverage, it is added into the *Queue* for the further mutation (Line 15). In addition, we further analyze its memory consumption. MemLock checks whether $child_i$ leads to more memory consumption based on $fmmap[k]$ and $ommap[k]$ at Line 16. (see Section 3.2.1). If so, MemLock updates the value of $fmmap[k]$ and $ommap[k]$ using the function *Update* at Line 17 (see Section 3.2.2). This process is repeated until the given time or resource budget expires (Lines 3).

3.2.1 Guidance Mechanisms. One of the most important components in the grey-box fuzzing is its guidance mechanism (Lines 14 and 16 in Algorithm 1), which often dominates the capability of the fuzzing technique in finding bugs [11, 33]. For example, SlowFuzz [53] uses the number of executed instructions as guidance to stress algorithmic complexity vulnerabilities. To find the memory consumption bugs effectively, MemLock uses branch coverage as well as memory consumption as the guidance. The branch coverage information guides MemLock to explore different program paths, while the memory consumption information can drive MemLock to focus on program paths with more memory consumption. To facilitate the description of our memory consumption guidance, we define the following concepts.

Definition 3.3 (Maximum Function Memory). *Given a path k and a set I of inputs that all execute k, the maximum function memory consumption $fmmap[k]$ in k is the maximum peak value of call stack, among all the inputs I:*

$$fmmap[k] \leftarrow \max_{i \in I} fm_i$$

where fm_i represents the peak value of call stack during the execution of input i (see Section 3.1.2).

Definition 3.4 (Maximum Operation Memory). *Given a path k and a set I of inputs that all execute k, the maximum operation memory consumption $ommap[k]$ in k is the maximum peak value of memory consumption by memory usage operations, among all the inputs I:*

$$ommap[k] \leftarrow \max_{i \in I} om_i$$

where om_i denotes the peak value of memory consumed by memory usage operations during the execution of input i (see Section 3.1.3).

Definition 3.5 (NewCov). *Given a set I of input and an input t, we say t hits a new coverage, if it either (1) executes a branch that has not been touched by I; or (2) hits a branch touched by I but with a different bucket number.*

The function *NewCov* (Line 14) will check whether a newly generated input $child_i$ hits a new coverage with respect the current *Queue* or not. That is, the function *NewCov* considers the branch coverage and guides MemLock to explore different program paths.

Definition 3.6 (NewMax). *Given a set I of input and an input t that all execute k, we say t hits a new maximum memory consumption, if either $fm_t > fmmap[k]$ or $om_t > ommap[k]$.*

The function *NewMax* (Line 16) determines whether the input $child_i$ leads to the maximum memory consumption among the current seed set. It actually checks two kinds of memory consumption. It first determines whether $child_i$ leads to the maximum function memory consumption (see Definition 3.3). It also considers whether $child_i$ leads to the maximum operation memory consumption (see Definition 3.4). If the input $child_i$ satisfies either of the above two cases, MemLock update the seed queue with $child_i$ at Line 17 (see Section 3.2.2).

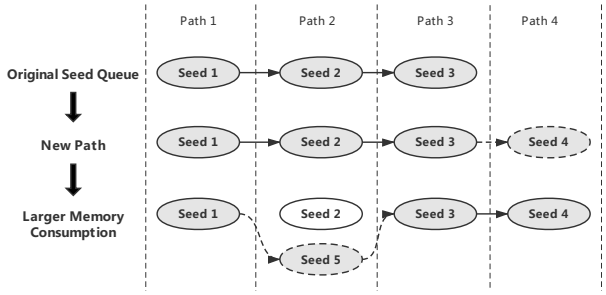

Figure 4: Dynamic Seed Updating

3.2.2 Dynamic Seed Updating. In order to efficiently support retaining the most interesting input for each path, we proposed a novel seed updating scheme. In MemLock, the seed queue is kept in a linked list, where each node represents a seed that explores a program path, as shown in Fig. 4. MemLock updates the seed queue in the following two cases. (1) *New Path.* If the test input results in new branch coverage, then it will be added to the seed queue as a new node, as shown in the second row of Fig. 4. (2) *Larger Memory Consumption.* If the input, e.g., *seed2* in the third row of Fig. 4, generates an input *seed5*, which does not result in new branch coverage, but it leads to larger memory consumption than the corresponding input. When *seed2* and *seed5* execute the same path, *seed2* is replaced with *seed5*. With replacing the original seed with the generated input $child_i$, we well exploit the advantage of $child_i$ as it is better in terms of finding memory consumption bugs. This seed updating policy ensures MemLock to gradually improve the overall memory consumption, and it could avoid getting stuck in local maxima like SlowFuzz [33], and brings long-term stable improvements.

To tailor for our guidance mechanism, MemLock also optimizes the seed selection probability (Line 5 in Algorithm 1) for the mutation as follows.

Definition 3.7 (Favored Input). *An input t is favored for mutation, if t has new branch coverage (i.e. NewCov) or t leads to maximum memory consumption (i.e., NewMax).*

Definition 3.8 (Selection Probability). *An input t is selected for mutation with the following probability:*

$$FuzzProb_t = \begin{cases} 1 & if\ t\ is\ favored \\ a & otherwise \end{cases}$$

That is, the favored inputs are always selected, and a is the probability of selecting a non-favored input. In our experiments we use $a = 0.01$ like PerfFuzz [33].

4 EVALUATION

We have built a prototype of MemLock. Our implementation adds around 1.6k lines of C/C++ code to the file containing AFL's core implementation. In particular, the static analysis and instrumentation components are implemented based on the LLVM framework [32], and the fuzzer engine is implemented based on the AFL-2.52b framework [76]. We have conducted thorough experiments to evaluate MemLock with a set of real-world programs. More detailed experimental results can be found on our website [68]. With these experiments, we aim to answer the following research questions:

RQ1. How capable is MemLock in memory consumption crash detection?

RQ2. How capable is MemLock in memory consumption real-world vulnerability detection?

RQ3. Do the strategies of MemLock help to trigger memory leaks with more leakage?

RQ4. Do the strategies of MemLock help to generate inputs with more memory consumption?

4.1 Experiment Setup

Following the suggestions in [31], we conducted the experiments carefully, to draw conclusions as objective as possible.

Baseline Fuzzers to Compare against. We compared MemLock against several state-of-the-art fuzzers, including AFL [76], AFLfast [8], PerfFuzz [33], FairFuzz [34] and QSYM [75]. The baseline fuzzers are selected based on the following considerations. AFL is the most popular baseline fuzzer studied in the community. AFLfast is an advanced variant of AFL, specially equipped with a better power schedule [8]. PerfFuzz [33] is to stress the time complexity issues in the program, while MemLock seeks to detect space complexity issues. Further, FairFuzz [34] leverages a targeted mutation strategy to execute towards rare branches. Lastly, QSYM [75] is a popular symbolic execution assisted fuzzer. In a word, we selected various kinds of representative state-of-the-art fuzzers as baseline fuzzers, and they are widely used to discover vulnerabilities in practice.

Evaluation Benchmarks. We select evaluation benchmarks considering several factors, e.g., popularity, frequency of being tested, development activeness, and functional diversity. Finally, we use 14 widely-used real-world programs, which all contain memory consumption bugs, to evaluate MemLock, including well-known development tools (e.g., *nm*, *cxxfilt*, *readelf*), code processing tools (e.g., *nasm*, *flex*, *yaml-cpp*, *mjs*), graphics processing libraries (e.g., *openjpeg*, *jasper*, *exiv2*), video processing tools (e.g., *bento4* and *libming*), and data processing libraries (e.g., *libsass* and *yara*), etc. These programs have also been widely tested by existing state-of-the-art greybox fuzzers [25, 31, 34, 74].

Performance Metrics. To compare against state-of-the-art fuzzers, the most direct measurement is the capability to find the vulnerabilities. With this regard, we consider both unique bugs and unique crashes each fuzzer finds in the fuzzing process. Since MemLock is to stress the space complexity issues of programs, we also distill the memory consumption of each seed in the pool.

Configuration Parameters. Since the fuzzers heavily rely on the random mutation, there could be performance jitter during fuzzing process. We took two actions to mitigate the randomness caused by the nature of fuzzing techniques. First, we test each program for a longer time, until the fuzzer reaches a relatively stable state. We run each fuzzer for 24 hours. Second, we perform each experiment for 5 times, and evaluate their statistical performance. Besides, we run all the fuzzers with the *-d* option to skip the deterministic mutation stage, following the configuration of PerfFuzz [33].

Memory Consumption Bugs. The uncontrolled-recursion bug usually causes stack-overflow, thus we can directly use AddressSanitizer [57] to detect it. The uncontrolled-memory-allocation bug consumes a large amount of memory so that the program runs

out of the memory. Thus, we can detect it by setting the "*allocator_may_return_null*" [26] flag of AddressSanitizer. In addition, we use LeakSanitizer [55] to detect memory leakage.

Experiment Infrastructure. All our experiments have been performed on machines with an Intel (R) Xeon (R) E5-1650 v3 Processor (3.40GHz) and 16GB of RAM under 64-bit Ubuntu LTS 16.04.

4.2 Unique Crashes Evaluation (RQ1)

To evaluate the effectiveness of fuzzers, a direct measurement is the number of unique crashes found by different fuzzers. It is believed that more unique crashes usually indicate higher chances of covering more unique vulnerabilities.

Table 1 shows the number of unique crashes, which is caused by memory consumption vulnerabilities, found by 6 different fuzzers within 24 hours in the benchmark programs. It is worth noting, we identify unique crashes related to memory consumption bugs by reproducing the crashes and analyzing their crash stacks. And we discuss other types of crashes in Section 4.6. Out of the 17 groups of experiments, MemLock performs best in 14 (82.4%) groups of experiments among 6 different fuzzers, as shown in column *MemLock*. In total, MemLock finds 2009 unique memory consumption crashes in the benchmark programs, improving by 59.2%, 70.5%, 76.9%, 98.1% and 66.7% respectively, compared to state-of-the-art fuzzers AFL, AFLfast, PerfFuzz, FairFuzz and QSYM. Especially, MemLock is able to find unique crashes in all benchmark programs, while other 5 state-of-the-art fuzzers may find no crashes in some benchmark programs. For example, other 5 state-of-the-art fuzzers cannot find any unique crashes in the program *flex*, however, MemLock can find 61 unique crashes within 24 hours. To better compare different fuzzers, we also use the plots to depict the performance over time in some benchmark programs, as shown in Figure 5. It shows that MemLock has a steady and strong growth trend in finding unique crashes, and MemLock is also the first fuzzer that reported crashes.

Following Klees' recommendation [31], we also conduct the statistic test for the results. The $\hat{A}_{12}$ [63] statistic measures the probability that one fuzzer outperforms another fuzzer, as shown in columns with the heading $\hat{A}_{12}$. Further, we use *Mann-Whitney U* [2] to measure the statistical significance of performance gain. When significant, we mark the corresponding $\hat{A}_{12}$ values in the bold. Out of 85 $\hat{A}_{12}$ values in the table, 60 (70.6%) $\hat{A}_{12}$ values are bold and exceeding the conventionally large effect size (0.71). Thus, we can conclude that MemLock significantly outperforms other 5 state-of-the-art fuzzers in most benchmark programs.

> From the analysis of Table 1 and Figure 5, we can positively answer **RQ1** that MemLock significantly outperforms the start-of-the-art fuzzers in terms of memory consumption crashes detection.

4.3 Real-world Vulnerability Evaluation (RQ2)

In this section, we compare the capability of MemLock to find real-world known vulnerabilities against baseline fuzzers, as suggested by Klees [31].

Table 2 shows the statistic results in 6 different state-of-the-art fuzzers. The benchmark programs totally contain 35 unique vulnerabilities, out of which MemLock performs best in the 27 vulnerabilities among 6 state-of-the-art fuzzers, as shown in column *MemLock*. MemLock averagely takes about 5.4 hours to find each unique vulnerability, which is 2.15, 2.15, 2.20, 2.69, 2.07 times faster than the state-of-the-art fuzzers AFL, AFlfast, PerfFuzz, FairFuzz, and QSYM respectively. In particular, MemLock finds 34 out of 35 unique vulnerabilities within 24 hours, while other fuzzers AFL, AFLfast, PerfFuzz, FairFuzz and QSYM only find 26, 28, 20, 17 and 25, respectively. The four unique vulnerabilities (i.e., issue#106, CVE-2018-18701, CVE-2019-6293 and CVE-2019-7698) in *mjs*, *nm*, *flex*, and *bento4* can be found only by MemLock within 24 hours. Therefore, it is proved that our memory-consumption guided strategy is very effective in finding memory consumption bugs.

In addition, we also conduct the statistic test for unique vulnerability evaluation. Out of 170 $\hat{A}_{12}$ values in the table, 111 (65.3%) $\hat{A}_{12}$ values are bold and exceeding the conventionally large effect size (0.71). Thus, MemLock significantly outperforms other 5 state-of-the-art fuzzers in finding unique vulnerabilities.

Case Study. To demonstrate the reason behind MemLock's superiority, we present the case of CVE-2019-6293. It is an uncontrolled-recursion vulnerability in *flex*, which is a lexical analyzer generator. The lexical analyzer generated by *flex* has to provide "beginning" state and "ending" states. The `mark_beginning_as_normal` function mark each "beginning" state in a machine as being a "normal" state, and the "beginning" states are the epsilon closure of the first state. The `mark_beginning_as_normal` function would call to itself if there is a state reachable from the first state through epsilon. We investigate MemLock's mutation history and identify a key mutation step. The test case triggers the `mark_beginning_as_normal` function calling itself for multiple times, through *havoc* mutation operation. Then, the recursive depth of this function is multiplied by *splice* operation, and finally leading to stack-overflow.

More interestingly, MemLock takes only 5.4 hours on average to discover this vulnerability, while other fuzzers all fail. We can also see the peak length of call stack of *flex* in Figure 6. AFL does not retain any seed over 5000 lengths, as those inputs do not increase coverage. Comparing to AFL, MemLock intentionally keeps seeds that increase the peak length of call stack, and finally triggering stack-overflow. This explains the reason why MemLock can find the vulnerability, while AFL can not detect it in all 5 runs.

New Vulnerabilities MemLock Found. With MemLock, we have discovered many previously unknown security-critical vulnerabilities. These vulnerabilities were not previously reported. We informed the maintainers, and Mitre assigned 15 CVEs. Among these 15 CVEs, 8 CVEs are uncontrolled-recursion vulnerabilities, 5 are vulnerabilities due to uncontrolled-memory-allocation issues, and 2 are about memory leak vulnerabilities. An attacker might leverage these vulnerabilities to launch an attack, by providing well-conceived inputs that trigger excessive memory consumption. The developers actively patched the vulnerabilities with our reports. At the time of writing, 12 of these vulnerabilities have been patched. We are confident that MemLock is effective and viable in practice.

> From the analysis of Table 2, case study and new vulnerabilities MemLock found, we can positively answer **RQ2** that MemLock significantly outperforms the state-of-the-art fuzzers in terms of real-world memory consumption vulnerability detection.

Table 1: Unique Crashes Evaluation

Program	Version	SLoC	Type	MemLock	AFL		AFLfast		PerfFuzz		FairFuzz		QSYM	
				#Crashes	#Crashes	$\hat{A}_{12}$	#Crashes	$\hat{A}_{12}$	#Crashes	$\hat{A}_{12}$	#Crashes	$\hat{A}_{12}$	#Crashes	$\hat{A}_{12}$
mjs [48]	1.20.1	40k	UR	114	36	**1.00**	31	**1.00**	88	**0.96**	12	**1.00**	30	**1.00**
cxxfilt [5]	2.31	1,757k	UR	448	373	**1.00**	304	**1.00**	401	**0.88**	39	**1.00**	327	**1.00**
nm [5]	2.31	1,757k	UR	127	12	**1.00**	21	**1.00**	17	**1.00**	0	**1.00**	20	**1.00**
nasm [49]	2.14.03	105k	UR	132	6	**1.00**	4	**1.00**	40	**1.00**	0	**1.00**	4	**1.00**
flex [24]	2.6.4	27k	UR	61	0	**1.00**	0	**1.00**	0	**1.00**	0	**1.00**	0	**1.00**
yaml-cpp [72]	0.6.2	58k	UR	4	0	**1.00**	1	**1.00**	3	0.56	0	**1.00**	0	**1.00**
libsass [39]	3.5.4	27k	UR	23	6	**1.00**	4	**1.00**	23	0.53	11	0.88	7	**1.00**
yara [73]	3.5.0	45k	UR	156	34	**1.00**	33	**1.00**	65	**0.94**	13	**1.00**	31	**1.00**
readelf [5]	2.28	1,844k	UA	273	104	**1.00**	110	**1.00**	54	**1.00**	181	**0.88**	114	**1.00**
exiv2 [22]	0.26	84k	UA	10	11	**0.14**	11	0.20	6	**0.90**	15	**0.00**	8	0.52
openjpeg [50]	2.3.0	243k	UA	16	8	0.80	5	**1.00**	0	**1.00**	7	0.46	5	0.80
bento4 [4]	1.5.1	78k	UA	5	2	**1.00**	2	**0.98**	2	**1.00**	1	**1.00**	1	**1.00**
			ML	145	78	**1.00**	72	**1.00**	61	**1.00**	125	**1.00**	74	**1.00**
libming [38]	0.4.8	92k	UA	18	20	0.40	18	0.60	17	0.62	20	**0.20**	16	0.80
			ML	264	336	**0.20**	324	**0.00**	324	**0.00**	371	**0.00**	354	**0.00**
jasper [28]	2.0.14	44k	UA	3	2	**0.84**	3	0.56	0	**1.00**	3	0.56	2	**0.92**
			ML	210	234	**0.08**	235	**0.08**	35	**1.00**	216	0.40	212	0.46
Total Unique Crashes (Improvement)				**2009**	**1262 (+59.2%)**		**1178 (+70.5%)**		**1136 (+76.9%)**		**1014 (+98.1%)**		**1205 (+66.7%)**	

* UR means the uncontrolled-recursion bug, UA means the uncontrolled-memory-allocation bug, and ML means the memory leak. We highlight the $\hat{A}_{12}$ values in the bold if its corresponding *Mann-Whitney U* test is significant.

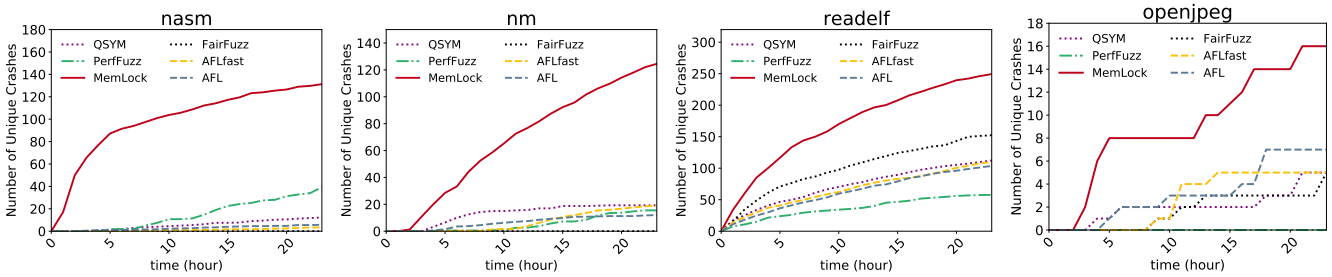

Figure 5: The growth trend of unique crashes found in different fuzzers; higher is better

4.4 Memory Leakage Evaluation (RQ3)

Memory leak bugs are a little different from uncontrolled-recursion and uncontrolled-memory-allocation bugs, because they may not lead to program crashes immediately. Only enough memory is leaked, it would produce Denial-of-Service (DoS) attack, for example, in a long time running programs (e.g., banking service). To evaluate the effectiveness of fuzzers in finding memory leaks, we look into the number of total bytes leaked within 24 hours during 6 different state-of-the-art fuzzers.

Table 3 shows the number of total bytes leaked for each fuzzer on different programs. We can see that MemLock shows an obvious advantage over other baseline fuzzers. The number of bytes leaked is improved (increased) by from 257% to 49633%, compared to other baseline fuzzers. This is because MemLock tries to maximize each allocation and generates inputs with high memory consumption. When the memory leak happens, those memory-consuming inputs will often cause more-bytes memory leakage.

From the results in Table 3, we can answer **RQ3** that MemLock significantly magnifies the memory leakage comparing to the state-of-the-art fuzzing techniques, due to its memory consumption guidance.

4.5 Memory Consumption Evaluation (RQ4)

Since MemLock seeks to generate test inputs that consume more and more memory. In this experiment, we evaluate the test input distribution according to memory consumption for each fuzzer we have tested. A fuzzer that maintains a seed pool with a larger proportion of high memory consumption inputs is considered to have a better chance of detecting memory consumption bugs.

Figure 6 shows the input distribution based on memory consumption. In general, we can clearly see that MemLock can generate more seeds with higher memory consumption. This is because the guidance mechanisms in MemLock help to gradually add more and more memory consuming inputs into the seed pool. In particular, for the uncontrolled-recursion bugs (*nm*, *nasm*, *flex* and *yara*), MemLock generates a large number of inputs that hold more than 30,000 function calls in the call stack, while PerfFuzz generates only a few and AFL/AFLfast can hardly generate inputs that hold more than 10,000 function calls. The pattern is similar for uncontrolled-memory-allocation bugs (*readelf*, *openjpeg*, *jasper* and *libming*). MemLock can generate a considerable amount of inputs with high memory consumption while the inputs of the other fuzzers concentrate on the low memory consumption region. The results clearly demonstrate the effectiveness of the strategies of MemLock in generating inputs with high memory consumption.

Table 2: Time to expose real-world vulnerability

Program	Vulnerability	Type	MemLock	AFL		AFLfast		PerfFuzz		FairFuzz		QSYM	
			Time(h)	Time(h)	$\hat{A}_{12}$	Time(h)	$\hat{A}_{12}$	Time(h)	$\hat{A}_{12}$	Time(h)	$\hat{A}_{12}$	Time(h)	$\hat{A}_{12}$
mjs	issue#58	UR	0.5	0.3	0.25	0.4	0.25	0.2	**0.13**	0.4	0.25	0.3	0.22
	issue#106	UR	13.7	T/O	**1.00**	T/O	**1.00**	T/O	**1.00**	T/O	**1.00**	T/O	**1.00**
cxxfilt	CVE-2018-9138	UR	0.3	7.2	**1.00**	10.1	**1.00**	0.5	**0.81**	T/O	**1.00**	3.3	**1.00**
	CVE-2018-9996	UR	T/O	16.5	**0.00**	T/O	0.50	T/O	0.50	T/O	0.50	T/O	0.50
	CVE-2018-17985	UR	0.2	1.1	**1.00**	4.5	**1.00**	0.2	0.63	1.9	**1.00**	1.4	**1.00**
	CVE-2018-18484	UR	0.2	1	**1.00**	4.5	**1.00**	0.2	0.63	8	**1.00**	1.4	**1.00**
	CVE-2018-18700	UR	0.2	1.2	**1.00**	4.6	**1.00**	0.3	**0.75**	12.6	**1.00**	1.4	**1.00**
nm	CVE-2018-12641	UR	2.6	19.1	**1.00**	12.6	**1.00**	12.2	**0.88**	T/O	**1.00**	12.8	**0.88**
	CVE-2018-17985	UR	10.4	18.2	**0.81**	11.9	0.56	T/O	**1.00**	T/O	**1.00**	13.3	0.63
	CVE-2018-18484	UR	9.9	16.4	**0.84**	17.1	**0.84**	T/O	**1.00**	T/O	**1.00**	14	0.75
	CVE-2018-18700	UR	9.6	14.9	0.63	17.8	**0.88**	T/O	**1.00**	T/O	**1.00**	T/O	**1.00**
	CVE-2018-18701	UR	13.9	T/O	**1.00**	T/O	**1.00**	T/O	**1.00**	T/O	**1.00**	T/O	**1.00**
	CVE-2019-9070	UR	18.4	15.6	0.56	13.9	0.44	T/O	**1.00**	T/O	**1.00**	15.8	0.56
	CVE-2019-9071	UR	12.4	T/O	**0.88**	14	0.69	T/O	**0.88**	T/O	**0.88**	T/O	**0.88**
nasm	CVE-2019-6290	UR	0.9	T/O	**1.00**	19	**1.00**	9	**1.00**	T/O	**1.00**	17.6	**1.00**
	CVE-2019-6291	UR	1.5	9	**0.94**	14	**1.00**	8.7	**1.00**	T/O	**1.00**	7.5	**1.00**
flex	CVE-2019-6293	UR	5.4	T/O	**1.00**	T/O	**1.00**	T/O	**1.00**	T/O	**1.00**	T/O	**1.00**
yaml-cpp	CVE-2019-6292	UR	0.4	T/O	**1.00**	18.4	**1.00**	0.9	**0.81**	T/O	**1.00**	T/O	**1.00**
	CVE-2018-20573	UR	6.1	T/O	**0.88**	T/O	**0.84**	12.4	**0.84**	T/O	**0.84**	T/O	**0.84**
libsass	CVE-2018-19837	UR	1.6	13.3	**0.88**	10.5	**0.88**	1.8	0.63	8.5	**0.88**	5	**0.81**
	CVE-2018-20821	UR	0.1	5.7	**1.00**	6.5	**1.00**	0.1	0.50	9.5	**1.00**	7.4	**1.00**
	CVE-2018-20822	UR	15.6	14.3	0.50	19.5	0.56	14.6	0.47	11.3	0.56	10.5	0.44
yara	CVE-2017-9438	UR	0.2	0.9	**1.00**	4.3	**1.00**	0.61	**0.91**	5.3	**1.00**	0.8	**1.00**
readelf	CVE-2017-15996	UA	0.2	0.3	**0.86**	0.2	0.68	0.5	**0.92**	0.3	0.68	0.3	**0.96**
exiv2	CVE-2018-4868	UA	0.1	0.1	0.50	0.1	0.50	0.1	0.50	0.1	0.50	0.1	0.50
bento4	CVE-2018-20186	UA	0.4	0.4	0.50	0.4	0.50	0.4	0.50	0.4	0.50	0.4	0.50
	CVE-2019-7698	UA	14.6	T/O	**1.00**	T/O	**1.00**	T/O	**1.00**	T/O	**1.00**	T/O	**1.00**
libming	CVE-2019-7581	UA	0.6	0.8	0.68	1.4	**0.80**	2	**0.88**	0.4	0.36	1.6	**0.80**
	CVE-2019-7582	UA	0.1	0.1	0.50	0.1	0.50	0.1	0.50	0.1	0.50	0.1	0.50
	issue#155	UA	1.4	1	0.30	1.3	0.36	1.4	0.40	1.2	0.42	1.6	0.64
openjpeg	CVE-2019-6988	UA	7.8	15.1	**0.86**	11.1	**0.84**	T/O	**1.00**	T/O	**1.00**	15.3	**0.81**
	CVE-2017-12982	UA	4.5	11.4	**0.72**	10	0.60	T/O	**1.00**	11.9	0.64	10	0.50
jasper	CVE-2016-8886	UA	4.1	17	**0.88**	22.3	**1.00**	T/O	**1.00**	10.3	0.52	18.2	**0.88**
	issue#207	UA	1.7	2.2	0.62	3.6	0.68	T/O	**1.00**	2.2	0.68	4	0.64
Average Time Usage (Improvement)			**5.4**	**11.6 (2.15×)**		**11.6 (2.15×)**		**11.9 (2.20×)**		**14.5 (2.69×)**		**11.2 (2.07×)**	
Unique Vulnerabilities (Improvement)			**34**	**26 (+30.8%)**		**28 (+21.4%)**		**20 (+70.0%)**		**17 (+100.0%)**		**25 (+36.0%)**	

* UR means the uncontrolled-recursion bug, UA means the uncontrolled-memory-allocation bug. T/O means the fuzzer can't find this vulnerability throughout 24 hours across 5 repetitions. When we calculate the average time usage, we replace T/O with 24 hours. We highlight the $\hat{A}_{12}$ in the bold if its corresponding *Mann-Whitney U* test is significant.

Table 3: Total Leak Bytes

Program	Type	Tool	leakge (Bytes)	Improve.	*p*-value	$\hat{A}_{12}$
bento4	memory leak	MemLock	**52,709,574**	-	-	-
		AFL	151,862	+34609%	0.0061	1.0
		AFLfast	1,233,255	+4174%	0.0061	1.0
		PerfFuzz	105,984	+49633%	0.0061	1.0
		FairFuzz	1,910,466	+2659%	0.0061	1.0
		QSYM	141,512	+37147%	0.0060	1.0
libming	memory leak	MemLock	**176,320,785**	-	-	-
		AFL	4,869,594	+3521%	0.0061	1.0
		AFLfast	2,535,212	+6855%	0.0061	1.0
		PerfFuzz	47,044,964	+257%	0.0061	1.0
		FairFuzz	828,742	+21176%	0.0061	1.0
		QSYM	1,219,093	+14363%	0.0061	1.0
jsaper	memory leak	MemLock	**2,372,844,732**	-	-	-
		AFL	56,018,839	+4136%	0.0061	1.0
		AFLfast	48,403,244	+4802%	0.0061	1.0
		PerfFuzz	6,229,898	+37988%	0.0061	1.0
		FairFuzz	56,788,235	+4096%	0.0061	1.0
		QSYM	38,244,568	+6104%	0.0061	1.0

After analyzing Figure 6, we can answer **RQ4** that the strategies of MemLock indeed help to generate inputs with great memory consumption.

4.6 Discussion

Additional Experiments. The above four groups of experiments show that MemLock is effective and efficient in finding memory consumption vulnerabilities. Since MemLock focuses on the space complexity issues, it may fall behind other baseline fuzzers in other performance metrics. For example, MemLock intentionally keeps seeds that increase memory consumption, which may degrade the capability to find other types of vulnerabilities. We have also evaluated the capability to find other types of crashes. In the benchmark programs, the state-of-the-art fuzzers MemLock, AFL, AFLfast, PerfFuzz, FairFuzz and QSYM find 77, 239, 228, 189, 276 and 236 other types of unique crashes, respectively. Moreover, our approach may also incur some runtime overhead. Therefore, we compare the code coverage and execution speed for each baseline fuzzer. In total, the number of executed test inputs in MemLock is from 20% to 84% of those in other fuzzers (AFL, AFLfast, FairFuzzer, QSYM). However, PerfFuzz performs worst among the six fuzzers, because PerfFuzz prefers the test inputs that execute long instructions. Considering the code coverage, MemLock achieves the comparable code coverage, compared to the other four fuzzers AFL, AFLfast, FairFuzzer and QSYM. PerfFuzz still performs the worst among the six fuzzers, and in most cases it only achieves the code coverage from about 60%

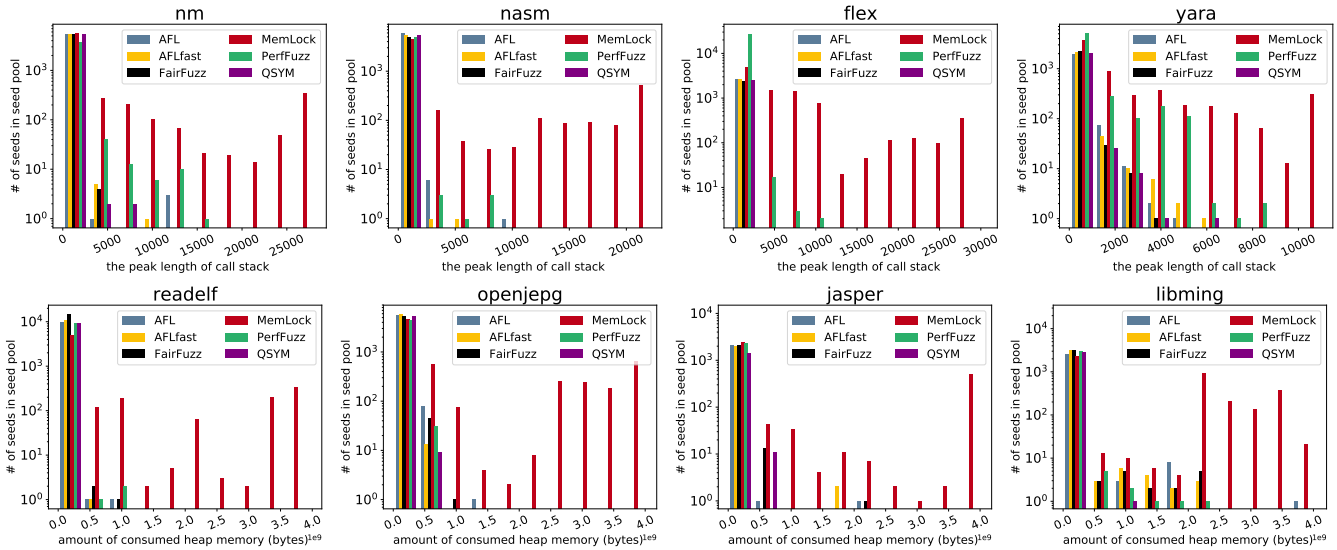

Figure 6: Seed distribution based on memory consumption. The larger the value on the right side is better.

to 70% of those in other fuzzers. These extra experimental results and data are available on our website[68] for interested readers.

Threats to Validity. We selected a variant of real-world programs to show the capabilities of MemLock, and compared it against other state-of-the-art fuzzers. However, our benchmark may still include a certain sample bias. Studies on more real-world programs can help better evaluate MemLock. Besides, MemLock also suffers from the difficulty in breaking through hard comparisons (e.g., magic bytes) as most work [7, 11, 25]. Adopting some program analysis techniques (e.g., symbolic execution), we can mitigate this threat.

5 RELATED WORK

Coverage-based Grey-box Fuzzing. Coverage-based grey-box fuzzing [3, 35, 37, 40, 43, 52, 61] is one of the most effective techniques to find vulnerabilities and bugs, and has attracted a great deal of attention from both academic and industry. Coverage-based grey-box fuzzers typically adopt the coverage information to guide different program path explorations. For example, Google has built an OSS-FUZZ platform [56] by incorporating several state-of-the-art coverage-based grey-box fuzzers: libFuzzer [41], honggfuzz [9], AFL [76] and ClusterFuzz [27].

Since coverage guidance engine is a key component for the grey-box fuzzers, much effort has been devoted to improve their coverage. Steelix [36], Vuzzer [54] and REDQUEEN [3] use program-state analysis or taint analysis to penetrate some paths protected by magic bytes comparisons. QSYM [75], Driller [59] and SAFL [67] equips grey-box fuzzing with a symbolic execution engine to reach deeper program code. Angora [12] adopts a gradient descent technique to solve path constraints so as to break some hard comparisons. ProFuzzer [74], GRIMOIRE [6], Superion [66] and Zest [51] leverage the knowledge in highly-structured files to generate syntactically and semantically valid test inputs, and thus be able to touch deeper program code. CollAFL [25] proposes a coverage sensitive fuzzing solution to mitigate the path collisions. FairFuzz [34] leverages a targeted mutation strategy to execute towards rare branches. Besides, AFLgo [7] and Hawkeye [11] use the distance metrics to execute towards user-specified target sites in the program. The main difference between MemLock and these state-of-the-art fuzzers is that, MemLock aims at memory consumption bugs while the others are to find memory corruption vulnerabilities. Thus, MemLock is orthogonal to these state-of-the-art fuzzers.

Recently, researchers have paid attention to the algorithmic complexity vulnerabilities (i.e., time complexity issues) such as SlowFuzz [53], Singularity [69] and PerfFuzz [33]. They use the number of executed instructions as the guidance to explore the program path with a longer path length. In contrast with MemLock, they stress the time complexity issues while MemLock considers space complexity issues. The space complexity issues have its own unique characteristics, as the amount of memory consumption can increase (e.g., function entry, memory allocation) and decrease (e.g., function exit, memory free), MemLock takes both of them into consideration.

Static Analysis. Static analysis is also used to analyze memory consumption [1, 10, 30, 65]. Wang *et al.* [65] presents a type-guided worst-case input generation by using automatic amortized resource analysis to derive symbolic bounds on the resource usage of functions. Duc-Hiep *et al.* [13] presents a worst-case memory consumption analysis, which uses symbolic execution to exhaustively unroll loops and compute memory consumption of each iteration. These approaches rely on type theory or symbolic execution, thus they often suffer from the scalability issue. SMOKE [23] is a path-sensitive memory leak detector for millions of lines of code. It first uses a scalable but imprecise analysis to compute a set of candidate memory leak paths and then verifies the feasibility of the candidates using a more precise analysis. While SMOKE can demonstrate the existence of memory leak, MemLock can generate an input that produces the memory leak.

Dynamic Analysis. Yuku *et al.* [42] proposes an improved real-time scheduling algorithm to reduce maximal heap memory consumption by controlling multitask scheduling. Different from MemLock, this technique aims at reducing memory consumption by dynamic online scheduling while MemLock is to find memory consumption bugs. BLEAK [64] is a system to debug memory leaks in

web applications. It leverages the observation that users often repeatedly return to the same visual state. Sustained growth between round trips is a strong indicator of a memory leak. BLEAK is only applicable to memory leak of web applications, while MemLock can find several kinds of memory consumption bugs. Radmin [21] is a system for early detection of application-level resource exhaustion and starvation attacks. It first learns and executes multiple probabilistic finite automata from its benign executions. It then restricts the resource usage to the learned automata and detects resource usage anomalies. Radmin uses some heuristics to detect resource usage anomalies, while MemLock employs the fuzzing technique to automatically generate the inputs for memory consumption bugs.

6 CONCLUSION

In this paper, we propose MemLock, an enhanced grey-box fuzzing technique to find memory consumption bugs. MemLock employs both coverage and memory consumption information to guide the fuzzing process. The coverage information guides the exploration of different program paths, while the memory consumption information guides the search for those program paths that exhibit more and more memory consumption. Our experimental results have shown that MemLock outperforms state-of-the-art fuzzing techniques (i.e., AFL, AFLfast, PerfFuzz, FairFuzz and QSYM) in detecting memory consumption bugs. We also found 15 security-critical vulnerabilities in some real-world programs. At the time of writing, 12 of these vulnerabilities have been patched.

ACKNOWLEDGEMENTS

The authors would like to thank the anonymous reviewers for their constructive comments. This work was partially supported by the National Natural Science Foundation of China under Grants No. 61772347 and 61972260, Guangdong Basic and Applied Basic Research Foundation under Grant No. 2019A1515011577.

REFERENCES

[1] Jeppe L Andersen, Mikkel Todberg, Andreas E Dalsgaard, and René Rydhof Hansen. 2013. Worst-case memory consumption analysis for SCJ. In *Proceedings of the 11th International Workshop on Java Technologies for Real-time and Embedded Systems*. ACM, 2–10.

[2] Andrea Arcuri and Lionel Briand. 2011. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *Software Engineering, 2011 33rd International Conference on*. IEEE, 1–10.

[3] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. 2019. REDQUEEN: Fuzzing with Input-to-State Correspondence. In *Proceedings of the Network and Distributed System Security Symposium*.

[4] Bento4. 2019. Full-featured MP4 format and MPEG DASH library and tools. http://www.bento4.com. accessed: 2019-08-01.

[5] GNU binutils. 2019. a collection of binary tools. https://www.gnu.org/software/binutils/. accessed: 2019-08-01.

[6] Tim Blazytko, Cornelius Aschermann, Moritz Schlögel, Ali Abbasi, Sergej Schumilo, Simon Wörner, and Thorsten Holz. 2019. GRIMOIRE: Synthesizing Structure while Fuzzing. (2019).

[7] 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, 2329–2344.

[8] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2017. Coverage-based greybox fuzzing as markov chain. *IEEE Transactions on Software Engineering* (2017).

[9] Maintained by Google. 2018. honggfuzz. http://honggfuzz.com/.

[10] Quentin Carbonneaux, Jan Hoffmann, Tahina Ramananandro, and Zhong Shao. 2014. End-to-end verification of stack-space bounds for C programs. In *ACM SIGPLAN Notices*, Vol. 49. ACM, 270–281.

[11] 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, 2095–2108.

[12] Peng Chen and Hao Chen. 2018. Angora: Efficient Fuzzing by Principled Search. *arXiv preprint arXiv:1803.01307* (2018).

[13] Duc-Hiep Chu, Joxan Jaffar, and Rasool Maghareh. 2016. Symbolic execution for memory consumption analysis. *ACM SIGPLAN Notices* 51, 5 (2016), 62–71.

[14] CVE-2017-9804. 2017. Available from MITRE. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-9804.

[15] CVE-2018-17985. 2018. Available from MITRE. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-17985.

[16] CVE-2018-4868. 2019. Available from MITRE. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-4868.

[17] CVE-2019-6291. 2019. Available from MITRE. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-6291.

[18] CVE-2019-6292. 2019. Available from MITRE. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-6292.

[19] CVE-2019-7704. 2019. Available from MITRE. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-7704.

[20] CVE Details. accessed: 2019. The list of Vulnerabilities according to CWE-400: Uncontrolled Resource Consumption. https://www.cvedetails.com/cwe-details/400/Uncontrolled-Resource-Consumption-039-Resource-Exhaustion.html.

[21] Mohamed Elsabagh, Daniel Barbará, Dan Fleck, and Angelos Stavrou. 2018. On early detection of application-level resource exhaustion and starvation. *Journal of Systems and Software* 137 (2018), 430–447.

[22] Exiv2. 2019. Image metadata library and tools. http://www.exiv2.org/. accessed: 2019-08-01.

[23] Gang Fan, Rongxin Wu, Qingkai Shi, Xiao Xiao, Jinguo Zhou Zhou, and Charles Zhang. 2019. SMOKE: Scalable Path-Sensitive Memory Leak Detection for Millions of Lines of Code. In *Proceedings of the 41st International Conference on Software Engineering, ICSE, Gothenburg, Sweden*.

[24] Flex. 2019. The Fast Lexical Analyzer - scanner generator for lexing in C and C++. https://github.com/westes/flex. accessed: 2019-08-01.

[25] 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, 679–696.

[26] Google. 2018. The list of common sanitizer options. https://github.com/google/sanitizers/wiki/SanitizerCommonFlags.

[27] Google. 2019. ClusterFuzz. https://google.github.io/clusterfuzz/.

[28] Jasper. 2019. Image Processing/Coding Tool Kit. https://www.ece.uvic.ca/~frodo/jasper/. accessed: 2019-08-01.

[29] Xiangkun Jia, Chao Zhang, Purui Su, Yi Yang, Huafeng Huang, and Dengguo Feng. 2017. Towards efficient heap overflow discovery. In *26th USENIX Security Symposium*. 989–1006.

[30] Daniel Kästner and Christian Ferdinand. 2014. Proving the absence of stack overflows. In *International Conference on Computer Safety, Reliability, and Security*. Springer, 202–213.

[31] 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.

[32] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*. IEEE Computer Society, 75.

[33] 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.

[34] 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.

[35] Jun Li, Bodong Zhao, and Chao Zhang. 2018. Fuzzing: a survey. *Cybersecurity* 1, 1 (2018), 6.

[36] 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*. ACM, 627–637.

[37] Hongliang Liang, Xiaoxiao Pei, Xiaodong Jia, Wuwei Shen, and Jian Zhang. 2018. Fuzzing: State of the art. *IEEE Transactions on Reliability* 67, 3 (2018), 1199–1218.

[38] Libming. 2019. A library for generating Macromedia Flash files. http://www.libming.org/. accessed: 2019-08-01.

[39] Libsass. 2019. A C/C++ implementation of a Sass compiler. https://github.com/sass/libsass. accessed: 2019-08-01.

[40] Xiaolong Liu, Qiang Wei, Qingxian Wang, Zheng Zhao, and Zhongxu Yin. 2018. CAFA: A Checksum-Aware Fuzzing Assistant Tool for Coverage Improvement. *Security and Communication Networks* (2018).

[41] LLVM-Documentation. 2018. libFuzzer - a library for coverage-guided fuzz testing. http://llvm.org/docs/LibFuzzer.html.

[42] Yuki Machigashira and Akio Nakata. 2018. An Improved LLF Scheduling for Reducing Maximum Heap Memory Consumption by Considering Laxity Time. In *2018 International Symposium on Theoretical Aspects of Software Engineering*. IEEE, 144–149.

[43] Valentin JM Manes, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J Schwartz, and Maverick Woo. 2018. Fuzzing: Art, Science, and Engineering. *arXiv preprint arXiv:1812.00140* (2018).

[44] MITRE. accessed: 2019. CWE-400: Uncontrolled Resource Consumption. https://cwe.mitre.org/data/definitions/400.html.

[45] MITRE. accessed: 2019. CWE-401: Missing Release of Memory after Effective Lifetime. https://cwe.mitre.org/data/definitions/401.html.

[46] MITRE. accessed: 2019. CWE-674: Uncontrolled Recursion. https://cwe.mitre.org/data/definitions/674.html.

[47] MITRE. accessed: 2019. CWE-789: Uncontrolled Memory Allocation. https://cwe.mitre.org/data/definitions/789.html.

[48] mjs. 2019. mjs: Restricted JavaScript engine. https://github.com/cesanta/mjs. accessed: 2019-08-01.

[49] Nasm. 2019. The Netwide Assembler. https://www.nasm.us. accessed: 2019-08-01.

[50] Openjpeg. 2019. An open-source JPEG 2000 codec written in C language. https://github.com/uclouvain/openjpeg. accessed: 2019-08-01.

[51] Rohan Padhye, Caroline Lemieux, Koushik Sen, Mike Papadakis, and Yves Le Traon. 2019. Semantic Fuzzing with Zest. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTAâĂŹ19)*.

[52] Hui Peng, Yan Shoshitaishvili, and Mathias Payer. 2018. T-Fuzz: fuzzing by program transformation. In *2018 IEEE Symposium on Security and Privacy*. IEEE, 697–710.

[53] 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, 2155–2168.

[54] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. 2017. Vuzzer: Application-aware evolutionary fuzzing. In *Proceedings of the Network and Distributed System Security Symposium*.

[55] Alexey Samsonov and Kostya Serebryany. 2013. New features in addresssanitizer. (2013).

[56] Kostya Serebryany. 2017. OSS-Fuzz-Google's continuous fuzzing service for open source software. (2017).

[57] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. AddressSanitizer: A fast address sanity checker. In *Presented as part of the 2012 USENIX Annual Technical Conference*. 309–318.

[58] Yuju Shen, Yanyan Jiang, Chang Xu, Ping Yu, Xiaoxing Ma, and Jian Lu. 2018. ReScue: crafting regular expression DoS attacks. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ACM, 225–235.

[59] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2016. Driller: Augmenting Fuzzing Through Selective Symbolic Execution.. In *NDSS*, Vol. 16. 1–16.

[60] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. 2013. Sok: Eternal war in memory. In *Security and Privacy, 2013 IEEE Symposium on*. IEEE, 48–62.

[61] Ari Takanen, Jared D Demott, Charles Miller, and Atte Kettunen. 2018. *Fuzzing for software security testing and quality assurance*. Artech House.

[62] Victor Van der Veen, Lorenzo Cavallaro, Herbert Bos, et al. 2012. Memory errors: The past, the present, and the future. In *International Workshop on Recent Advances in Intrusion Detection*. Springer, 86–106.

[63] András Vargha and Harold D Delaney. 2000. A critique and improvement of the CL common language effect size statistics of McGraw and Wong. *Journal of Educational and Behavioral Statistics* 25, 2 (2000), 101–132.

[64] John Vilk and Emery D Berger. 2018. BLeak: automatically debugging memory leaks in web applications. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 15–29.

[65] Di Wang and Jan Hoffmann. 2019. Type-Guided Worst-Case Input Generation. *Proceedings of the ACM on Programming Languages* (2019).

[66] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2019. Superion: Grammar-Aware Greybox Fuzzing. In *Proceedings of the 41st International Conference on Software Engineering, ICSE, Gothenburg, Sweden*.

[67] Mingzhe Wang, Jie Liang, Yuanliang Chen, Yu Jiang, Xun Jiao, Han Liu, Xibin Zhao, and Jiaguang Sun. 2018. SAFL: increasing and accelerating testing coverage with symbolic execution and guided fuzzing. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*. ACM, 61–64.

[68] Anonymous Website. accessed: 2019-08-01. MemLock. https://researchreview.github.io/MemLock/.

[69] Jiayi Wei, Jia Chen, Yu Feng, Kostas Ferles, and Isil Dillig. 2018. Singularity: Pattern fuzzing for worst case complexity. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 213–223.

[70] Technical whitepaper for afl fuzz. 2019. american fuzzy lop. http://lcamtuf.coredump.cx/afl/technical_details.txt. accessed: 2019-08-01.

[71] Zhiwu Xu, Cheng Wen, and Shengchao Qin. 2018. State-taint analysis for detecting resource bugs. *Science of Computer Programming* 162 (2018), 93–109.

[72] yaml cpp. 2019. A YAML parser and emitter in C++. https://github.com/jbeder/yaml-cpp. accessed: 2019-08-01.

[73] Yara. 2019. The pattern matching swiss knife for malware researchers. http://virustotal.github.io/yara/. accessed: 2019-08-01.

[74] 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 *Security and Privacy, 2019 IEEE Symposium on*. IEEE.

[75] 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.

[76] Michal Zalewski. 2017. American Fuzzy Lop 2.52b. http://lcamtuf.coredump.cx/afl/.