MemLock: Memory Usage Guided Fuzzing

Cheng Wen
CSSE, Shenzhen University
Shenzhen, China

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

Yuekang Li
Nanyang Technological University
Singapore

Shengchao Qin∗
SCE, 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, AFL fast, PerfFuzz, FairFuzz, Angora and QSYM, in discovering memory consumption bugs. During the experiments, we discovered many previously unknown memory consumption bugs and received 15 new CVEs.

CCS CONCEPTS
• Security and privacy → Software security engineering.

KEYWORDS

Fuzz Testing, Software Vulnerability, Memory Consumption

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., [17–23, 74]). For example, if the termination conditions of recursive functions are not implemented correctly, an infinite number of recursive function calls can occur and thus render the stack memory exhausted. The adversaries can exploit this vulnerability to launch a Denial-of-Service (DoS) attack with some well-crafted inputs [18, 21]. Recently, researchers have started to pay attention to these issues. For example, SlowFuzz [58], PerfFuzz [37] and ReScue [63] 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.

Although a number of works (e.g., the popular fuzzing techniques [11, 28, 45, 61, 84]) 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 [65, 67, 72]. 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 [49], 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 [52], uncontrolled-memory-allocation [51], and memory leak [50]. 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 [15, 24, 46, 70, 79]. For example, Radmin [24] 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 greybox fuzzing [84] 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 [39, 41], 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 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 six state-of-the-art tools (i.e., AFL [84], AFLfast [8], PerfFuzz [37], FairFuzz [38], Angora [12] and QSYM [83]), in discovering the memory consumption vulnerabilities. MemLock finds 40.5% more unique crashes and 17.9% more vulnerabilities, than the second best counterpart. In particular, MemLock can discover a certain memory consumption vulnerability at least 2.07 times faster than the other baseline fuzzers. Besides, the generated test cases in MemLock usually lead to 150 times memory consumption compared to the other state-of-the-art tools. In addition, we have responsibly disclosed several previously unknown memory consumption bugs, and received 15 new CVE 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.

---

1 The Common Vulnerabilities and Exposures (CVE) system provides a reference for tracking publicly known information-security vulnerabilities and exposures.
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 [18] in Figure 1 to demonstrate an uncontrolled-recursion bug and CVE-2018-4868 [19] 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 [71] 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. 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 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 i1 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 have not 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. To facilitate the description of our methodology, we define the following concepts.

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 [84]. 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. Consequently, the branch coverage information in an executed program path can be defined as follows.

**Definition 3.1 (Trace Bits [84]).** For an executed program path, its trace bits are represented by an 8-bit array with size \( 2^K \), and the value of the IDth 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.

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

\footnote{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 [78].}
The value coverage-based grey-box fuzzers (e.g., AFL), the main difference is input. While this process is similar to the process of traditional memory (see Definitions 3.3 and 3.4), it is retained as an interesting executions. If a child input has new coverage or consumes more puts. Then, MemLock decides whether to mutate it based on a selection prob-

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

\[
\begin{align*}
S & \leftarrow \Phi; \\
Queue & \leftarrow T; \\
\text{while time and resource budget do not expire do} \\
\quad \text{for each input} \ t \ \text{in Queue do} \\
\quad \quad \text{if with probability} \ \text{FuzzProb}_t \ \text{to select} \ t \ \text{then} \\
\quad \quad \quad \text{numChildren} \leftarrow \text{AssignEnergy}(t); \\
\quad \quad \quad \text{for} \ 0 \leq i < \text{numChildren} \ \text{do} \\
\quad \quad \quad \quad \text{child}_i \leftarrow \text{Mutate}(t); \\
\quad \quad \quad \quad \text{traceBits}_i, \text{fm}_i, \text{om}_i \leftarrow \text{Run(child}_i, P); \\
\quad \quad \quad \quad \text{if} \ \text{it triggers memory consumption bugs then} \\
\quad \quad \quad \quad \quad S \leftarrow S \cup \text{child}_i; \\
\quad \quad \quad \quad \text{else} \\
\quad \quad \quad \quad \quad \text{if} \ \text{NewCov(traceBits}_i) \ \text{then} \\
\quad \quad \quad \quad \quad \quad \text{Queue} \leftarrow \text{Queue} \cup \text{child}_i; \\
\quad \quad \quad \quad \quad \text{if} \ \text{NewMax(fm}_i, \text{om}_i) \ \text{then} \\
\quad \quad \quad \quad \quad \quad \text{Queue} \leftarrow \text{Update(child}_i, \text{fmMap}[k], \text{omMap}[k]); \\
\quad \quad \text{end if} \\
\quad \quad \text{end if} \\
\quad \text{end for} \\
\text{end if} \\
\text{end for} \\
\text{return} S
\end{align*}
\]

its parameters and return value. The reason is that the memory is allocated by some standard library functions [1, 46], e.g., malloc, calloc, realloc, and free. 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 [33], 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 \( or \) bytes memory, the value \( ot \) is increased by \( or \); likewise, if it frees \( or \) bytes memory, the value \( ot \) is decreased by \( or \). 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.

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., \( \text{numChildren} \)) to it at Line 6, which determines the number of children to produce from \( t \). MemLock uses the same heuristics to determine \( \text{numChildren} \) as AFL [84]. 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 \( \text{numChildren} \) children, monitors their executions, and determines their affiliations. MemLock first performs mutation to generate the new input \( \text{child}_i \) (Line 8). At Line 9, MemLock then runs the input \( \text{child}_i \) on the instrumented program \( P \), and collects its branch coverage (i.e., \( \text{traceBits}_i \)), function memory consumption (i.e., \( \text{fm}_i \)), and operation memory consumption (i.e., \( \text{om}_i \)), respectively.

If the input \( \text{child}_i \) 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 \( \text{child}_i \) leads to more memory consumption based on \( \text{fmMap}[k] \) and \( \text{omMap}[k] \) at Line 16. (see Section 3.2.1). If so, MemLock updates the value of \( \text{fmMap}[k] \) and \( \text{omMap}[k] \) using the function \( \text{Update} \) 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, 37]. For example, SlowFuzz [58] 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 \( \text{fmMap}[k] \) in \( k \) is the maximum peak value of call stack, among all the inputs \( i \):

\[
\text{fmMap}[k] \leftarrow \max_{i \in I} \text{fm}_i
\]

where \( \text{fm}_i \) represents the peak value of call stack during the execution of input \( i \) (see Section 3.1.2).
**Figure 4: Dynamic Seed Updating**

**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 \( i \):

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

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 inputs and an input \( i \), we say \( i \) 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 \( \text{NewCov} \) (Line 14) will check whether a newly generated input \( \text{child}_i \) hits a new coverage with respect the current \( \text{Queue} \) or not. That is, the function \( \text{NewCov} \) considers the branch coverage and guides \( \text{MemLock} \) to explore different program paths.

**Definition 3.6 (NewMax).** Given a set \( I \) of inputs and an input \( i \) that all execute \( k \), we say \( i \) hits a new maximum memory consumption, if either \( fm_i > fmmap[k] \) or \( om_i > ommap[k] \).

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

### 3.2.2 Dynamic Seed Updating

In order to efficiently support retaining the most interesting input for each path, we propose a novel seed updating scheme. In \( \text{MemLock} \), 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. \( \text{MemLock} \) updates the seed queue in the following two cases. (1) \( \text{New Path} \). 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) \( \text{Larger Memory Consumption} \). If the input, e.g., \( \text{seed}_2 \) in the third row of Fig. 4, generates an input \( \text{seed}_5 \) which does not result in new branch coverage, but it leads to larger memory consumption than the corresponding input. When \( \text{seed}_2 \) and \( \text{seed}_5 \) execute the same path, \( \text{seed}_2 \) is replaced with \( \text{seed}_5 \). With replacing the original seed with the generated input \( \text{child}_i \), we well exploit the advantage of \( \text{child}_i \) as it is better in terms of finding memory consumption bugs. This seed updating policy ensures \( \text{MemLock} \) to gradually improve/increase the overall memory consumption, and it could avoid getting stuck in local maxima like SlowFuzz [37], and brings long-term stable improvements.

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

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

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

\[
\text{FuzzProb}_i = \begin{cases} 
1 & \text{if } i \text{ is favored} \\
\alpha & \text{otherwise}
\end{cases}
\]

That is, the favored inputs are always selected, and \( \alpha \) is the probability of selecting a non-favored input. In our experiments we use \( \alpha = 0.01 \) like PerfFuzz [37].

### 4 EVALUATION

We have built a prototype of \( \text{MemLock} \). 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 [36], and the fuzzer engine is implemented based on the AFL-2.52b framework [84]. We have conducted thorough experiments to evaluate \( \text{MemLock} \) with a set of real-world programs. More detailed experimental results can be found on our website [48]. With these experiments, we aim to answer the following research questions:

**RQ1.** How capable is \( \text{MemLock} \) in memory consumption crash detection?

**RQ2.** How capable is \( \text{MemLock} \) in memory consumption real-world vulnerability detection?

**RQ3.** Do the strategies of \( \text{MemLock} \) help to trigger memory leaks with more leakage?

**RQ4.** Do the strategies of \( \text{MemLock} \) help to generate inputs with more memory consumption?

#### 4.1 Experiment Setup

Following the suggestions in [35], we conducted the experiments carefully, to draw conclusions as objective as possible.

**Baseline Fuzzers to Compare against.** We compare \( \text{MemLock} \) against six state-of-the-art fuzzers, namely AFL [84], AFLfast [8], PerfFuzz [37], FairFuzz [38], Angora [12] and QSYM [83]. The baseline fuzzers are selected based on the following considerations. AFL is the widely-used coverage-based greybox fuzzer, and selected as baseline fuzzer in the most work. AFLfast is an advanced variant of AFL, specially equipped with a better power schedule [8]. PerfFuzz [37] is to stress the time complexity issues in the program, while \( \text{MemLock} \) seeks to detect space complexity issues. FairFuzz [38] leverages a targeted mutation strategy to execute towards rare branches. Further, Angora [12] utilizes taint analysis to track information flow, and then uses gradient descent to
break through the hard branches. Lastly, QSYM [83] is a popular symbolic execution assisted fuzzer. Note that we haven’t selected MemFuzz [16] as baseline fuzzer, because MemFuzz is not open source and it resorts to memory accesses (instead of memory consumption). 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., libass and yara), etc. These programs have also been widely tested by existing state-of-the-art fuzzers.

MemLOCK performs best in 10 (58.8%) groups of experiments among 7 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%, 40.5% and 66.7% respectively, compared to state-of-the-art fuzzers AFL, AFLfast, PerfFuzz, FairFuzz, Angora and QSYM. Especially, MemLOCK is able to find unique crashes in all benchmark programs, while other 6 state-of-the-art fuzzers may find no crashes in some benchmark programs. For example, none of the other 6 state-of-the-art fuzzers could find any unique crashes in the program flex, but MemLOCK was able to 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 [35], we also conduct the statistic test for the results. The $A_{12}$ [68] statistic measures the probability that one fuzzer (in this case MemLOCK) outperforms another fuzzer. The value of $A_{12}$ means by what chance the result of MemLOCK is better than the competitor, as shown in columns with the heading $A_{12}$. Further, we apply the Mann-Whitney U-test [2] with a significance level of 0.05 to check the statistical significance differences of experimental results. A smaller statistical significance difference (a.k.a $p$-value) indicates a more significant difference between MemLOCK and the competitor. In Table 1, we mark the corresponding $A_{12}$ values in bold for those with a $p$-value smaller than the significance level (0.05) for simplicity, we do not include $p$-values here but they are available at the companion website [48]). Out of 102 $A_{12}$ values in the table, 72 (70.6%) $A_{12}$ values exceed the conventionally large effect size (0.71) and are marked in bold. Thus, we can conclude that MemLOCK significantly outperforms other 6 state-of-the-art fuzzers in most benchmark programs.

**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 [35].

Table 2 shows the statistic results in MemLOCK as well as other 6 different state-of-the-art fuzzers. The benchmark programs totally contain 34 unique vulnerabilities, out of which MemLOCK performs best in the 25 vulnerabilities among other 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, 3.76, 2.07 times faster than the state-of-the-art fuzzers AFL, AFLfast, PerfFuzz, FairFuzz, Angora and QSYM respectively. In particular, MemLOCK finds 33 out of 34 unique vulnerabilities within 24 hours, while other fuzzers AFL, AFLfast, PerfFuzz, FairFuzz, Angora and QSYM only find 26, 28, 20, 17, 6 and 25, respectively. The three unique vulnerabilities (i.e., issue#106, CVE-2018-18701 and CVE-2019-6293) in mjs, nm and flex can be found only by
## Table 1: Unique Crashes Evaluation

<table>
<thead>
<tr>
<th>Program</th>
<th>Version</th>
<th>SLoC</th>
<th>Type</th>
<th>MemLock</th>
<th>AFL</th>
<th>AFLfast</th>
<th>PerfFuzz</th>
<th>FairFuzz</th>
<th>Angora</th>
<th>QSYM</th>
</tr>
</thead>
<tbody>
<tr>
<td>mjs [5]</td>
<td>1.20.1</td>
<td>40k</td>
<td>UR</td>
<td>114</td>
<td>36</td>
<td>1.00</td>
<td>88</td>
<td>12</td>
<td>0</td>
<td>1.00</td>
</tr>
<tr>
<td>exclud [5]</td>
<td>2.31</td>
<td>1,757k</td>
<td>UR</td>
<td>448</td>
<td>373</td>
<td>1.00</td>
<td>401</td>
<td>39</td>
<td>0</td>
<td>1.00</td>
</tr>
<tr>
<td>nm [5]</td>
<td>2.31</td>
<td>1,757k</td>
<td>UR</td>
<td>127</td>
<td>12</td>
<td>1.00</td>
<td>17</td>
<td>0</td>
<td>0</td>
<td>1.00</td>
</tr>
<tr>
<td>nasm [54]</td>
<td>2.14.03</td>
<td>105k</td>
<td>UR</td>
<td>132</td>
<td>6</td>
<td>1.00</td>
<td>40</td>
<td>0</td>
<td>0</td>
<td>1.00</td>
</tr>
<tr>
<td>flex [27]</td>
<td>2.6.4</td>
<td>27k</td>
<td>UR</td>
<td>61</td>
<td>0</td>
<td>1.00</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1.00</td>
</tr>
<tr>
<td>yaml-cpp [80]</td>
<td>0.6.2</td>
<td>58k</td>
<td>UR</td>
<td>4</td>
<td>0</td>
<td>1.00</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>1.00</td>
</tr>
<tr>
<td>libass [43]</td>
<td>3.5.4</td>
<td>27k</td>
<td>UR</td>
<td>23</td>
<td>6</td>
<td>1.00</td>
<td>23</td>
<td>11</td>
<td>0.88</td>
<td>1.00</td>
</tr>
<tr>
<td>yara [81]</td>
<td>3.5.0</td>
<td>45k</td>
<td>UR</td>
<td>156</td>
<td>34</td>
<td>1.00</td>
<td>65</td>
<td>13</td>
<td>0</td>
<td>1.00</td>
</tr>
<tr>
<td>readeif [5]</td>
<td>2.28</td>
<td>1,844k</td>
<td>UA</td>
<td>273</td>
<td>104</td>
<td>1.00</td>
<td>54</td>
<td>181</td>
<td>0.88</td>
<td>1.00</td>
</tr>
<tr>
<td>exiv2 [25]</td>
<td>0.26</td>
<td>84k</td>
<td>UA</td>
<td>10</td>
<td>11</td>
<td>0.14</td>
<td>6</td>
<td>0.90</td>
<td>15</td>
<td>0.00</td>
</tr>
<tr>
<td>openpeg [55]</td>
<td>2.30</td>
<td>243k</td>
<td>UA</td>
<td>16</td>
<td>8</td>
<td>0.80</td>
<td>5</td>
<td>0.46</td>
<td>0</td>
<td>1.00</td>
</tr>
<tr>
<td>bentot [4]</td>
<td>1.51</td>
<td>78k</td>
<td>UA</td>
<td>5</td>
<td>2</td>
<td>1.00</td>
<td>2</td>
<td>1.00</td>
<td>0</td>
<td>1.00</td>
</tr>
<tr>
<td>libmpg [42]</td>
<td>0.48</td>
<td>92k</td>
<td>UA</td>
<td>18</td>
<td>20</td>
<td>0.40</td>
<td>18</td>
<td>2</td>
<td>20</td>
<td>0.20</td>
</tr>
<tr>
<td>jasper [32]</td>
<td>2.0.14</td>
<td>44k</td>
<td>UA</td>
<td>3</td>
<td>2</td>
<td>0.84</td>
<td>3</td>
<td>0.56</td>
<td>0</td>
<td>1.00</td>
</tr>
<tr>
<td>readelf [5]</td>
<td>2.28</td>
<td>1,844k</td>
<td>UA</td>
<td>273</td>
<td>104</td>
<td>1.00</td>
<td>54</td>
<td>181</td>
<td>0.88</td>
<td>1.00</td>
</tr>
<tr>
<td>exiv2 [25]</td>
<td>0.26</td>
<td>84k</td>
<td>UA</td>
<td>10</td>
<td>11</td>
<td>0.14</td>
<td>6</td>
<td>0.90</td>
<td>15</td>
<td>0.00</td>
</tr>
<tr>
<td>openpeg [55]</td>
<td>2.30</td>
<td>243k</td>
<td>UA</td>
<td>16</td>
<td>8</td>
<td>0.80</td>
<td>5</td>
<td>0.46</td>
<td>0</td>
<td>1.00</td>
</tr>
</tbody>
</table>

**Total Unique Crashes (Improvement)**

1. UR means the uncontrolled-recursion bug, UA means the uncontrolled-memory-allocation bug, and ML means the memory leak. We highlight the A12 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](image)

**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 function beginning as normal function mark each “beginning” state in a machine as being a “normal” state, and the “beginning” states are the epison closure of the first state. The function 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 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.

**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. Detailed information on our newly discovered vulnerabilities is available on our website [48]. We are confident that MemLock is effective and viable in practice.
From the analysis of Table 2, the 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.

### 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 during 7 different fuzzers within 24 hours.

Table 3 shows the amount of memory leak (in bytes) identified by each fuzzer that may occur in 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 234% to 3753163%, 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.

### 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 MemLock, AFL, AFLfast, PerfFuzz, FairFuzz, Angora and QSYM. 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.
Among all the fuzzers, PerfFuzz performs the worst likely due to the strate-
gies of MEMLOCK indeed help to generate inputs with high 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 fail behind other baseline fuzzers in other performance metrics. For example, MEMLOCK intentionally keeps seeds that increase memory consumption, which may degrade its capability of identifying other types of vulnerabilities. We have therefore evaluated the capability of finding other types of crashes. In the benchmark programs, MEMLOCK, AFL, AFLFast, PerfFuzz, FairFuzzer, and QSYM find 77, 239, 228, 189, 276, 343 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 ranges from 20% to 84% of those in AFL, AFLFast, FairFuzzer and QSYM. Among all the fuzzers, PerfFuzz performs the worst likely due to the fact that it prefers the test inputs that execute long instructions. Considering the code coverage, MEMLOCK achieves the comparable code coverage, compared to the fuzzers AFL, AFLFast, FairFuzzer and QSYM. PerfFuzz still performs the worst among those fuzzers, and in most cases it only achieves the code coverage from about 60% to 70% of those in other fuzzers. All extra experimental results and data are available on our website[48] 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 benchmarks may still include
a certain sample bias. Further 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, 28]. Adopting some program analysis techniques (e.g., symbolic execution) might help mitigate this threat.

5 RELATED WORK

Coverage-based Grey-box Fuzzing. Coverage-based grey-box fuzzing [3, 39, 41, 44, 47, 57, 66] 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 [61] by incorporating several state-of-the-art coverage-based grey-box fuzzers: libFuzzer [45], honggfuzz [9], AFL [84] and ClusterFuzz [30].

Since a coverage guidance engine is a key component for the grey-box fuzzers, much effort has been devoted to improve their coverage. Steelix [40], Vuzzer [59] and REDQUEEN [3] use program-state analysis or taint analysis to penetrate some paths protected by magic bytes comparisons. QSYM [83], Driller [64] and SAFL [76] 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. MemFuzz [16] augmenting evolutionary fuzzing by additionally leveraging information about memory accesses (instead of memory consumption) performed by the target program. ProfFuzzer [82], GRIMOIRE [6], Superion [75] and Zest [56] leverage the knowledge in highly-structured files to generate syntactically and semantically valid test inputs, and thus be able to touch deeper program code. CoIIAFL [28] proposes a coverage sensitive fuzzing solution to mitigate the path collisions. FairFuzz [38] leverages a targeted mutation strategy to execute towards rare branches. UAFL [73] incorporates typestate properties and information flow to their fuzzing engine to guide the detection of use-after-free vulnerabilities. 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 [58], Singularity [77] and PerFuzz [37]. 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, 13, 14, 31, 34, 70]. Wang et al. [70] 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. [15] presents a worst-case memory consumption analysis, which uses symbolic execution to exhaustively unroll loops and compute memory consumption of each iteration. He et al. [31] and Chin et al. [14] employ static verification to check a program’s memory usage is within the memory bounds, while Chin et al. [13] uses static analysis to compute the memory usage bounds for assembly level programs. These approaches rely on type theory or symbolic execution, thus they often suffer from the scalability issue. SMOKE [26] 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. [46] 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 [69] 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 [24] 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, PerFuzz, FairFuzz, Angora 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

This work was supported in part by the National Natural Science Foundation of China under Grants No. 61772347, 61836005, 61972260, 61727408, 61721002, Ant Financial Services Group through Ant Financial Research Program, Guangdong Basic and Applied Basic Research Foundation under Grant No. 2019A1515011577, National Key R&D Program of China under Grant No. 2018YFB0803501.
MEMLOCK: Memory Usage Guided Fuzzing

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


