*mathematics*

MDPI

*Article*

# MemConFuzz: Memory Consumption Guided Fuzzing with Data Flow Analysis

**Chunlai Du [1], Zhijian Cui [1], Yanhui Guo [2,*], Guizhi Xu [1] and Zhongru Wang [1,3]**

[1] School of Information Science and Technology, North China University of Technology, Beijing 100144, China
[2] Department of Computer Science, University of Illinois Springfield, Springfield, IL 62703, USA
[3] Chinese Academy of Cyberspace Studies, Beijing 100048, China
* Correspondence: yguo56@uis.edu

**Abstract:** Uncontrolled heap memory consumption, a kind of critical software vulnerability, is utilized by attackers to consume a large amount of heap memory and consequently trigger crashes. There have been few works on the vulnerability fuzzing of heap consumption. Most of them, such as MemLock and PerfFuzz, have failed to consider the influence of data flow. We proposed a heap memory consumption guided fuzzing model named MemConFuzz. It extracts the locations of heap operations and data-dependent functions through static data flow analysis. Based on the data dependency, we proposed a seed selection algorithm in fuzzing to assign more energy to the samples with higher priority scores. The experiment results showed that the MemConFuzz has advantages over AFL, MemLock, and PerfFuzz with more quantity and less time consumption in exploiting the vulnerability of heap memory consumption.

**Keywords:** fuzzing; memory consumption; data flow; taint analysis

**MSC:** 90C70



**Citation:** Du, C.; Cui, Z.; Guo, Y.; Xu, G.; Wang, Z. MemConFuzz: Memory Consumption Guided Fuzzing with Data Flow Analysis. *Mathematics* **2023**, *11*, 1222. https://doi.org/10.3390/math11051222

Academic Editor: Ivan Lorencin

Received: 30 January 2023
Revised: 23 February 2023
Accepted: 27 February 2023
Published: 2 March 2023



## 1. Introduction

Fuzzing is a kind of random testing technique and is widely used to discover vulnerabilities in computer programs. Blind samples mutation fuzzing models and coverage-guided fuzzing models fail to select interesting seeds and waste testing time. Many fuzzing models are currently guided by exploring ways to improve path coverage. It is believed that the more code blocks that can be covered, the more likely potential vulnerability will be triggered. Many state-of-the-art fuzzing models typically use information from the programs' control flow graph by the program under test (PUT) to determine which samples would be selected as seeds for further mutation. Although there has been a lot of research work on memory overflow vulnerability, most of these methods have mainly exploited memory corruption vulnerabilities, such as stack buffer overflow, use-after-free (UAF), out-of-bounds reading, and out-of-bounds writing, etc. Memory corruption occurs when the contents of memory are overwritten due to malicious instructions or normal instructions with unexpected data beyond the program's original intention. For example, a buffer overflow occurs when a program tries to copy data into a variable whose required memory length is larger than the target. When corrupted memory contents are later used, the program triggers a crash or turns into a shellcode. Most fuzzing models of memory corruption vulnerability depend on the control flow, and seldom on the data semantics.

Memory consumption is a different kind of memory vulnerability in contrast to memory corruption, which is more like a logical vulnerability potentially existing in the action sequence of memory allocation and deallocation. With one goal of making more efficient use of the memory, different code segments in general are stored in different memory areas, among which the stack area and heap area are the two most important types of memory areas. In the process of a program running, the stack area grows up or down

by calling subfunctions. It contains local variables, stack register ebp of parent function, return address, and parameters from the parent function. Generally, the heap areas are a series of memory blocks allocated and freed by the programs, which can be used by the pointer of heap blocks. Memory consumption occurs in the process of heap allocation and release. When a program triggers instructions for heap memory allocation enough times without deallocating unused memory in time, it would likely lead to a crash. Uncontrolled heap memory consumption is therefore a critical issue of software security, and can also become an important vulnerability when attackers control execution flow to consume large amounts of memory, and thus, launch denial-of-service attacks.

To solve the problems in vulnerability fuzzing of heap consumption, we propose a heap memory consumption-guided fuzzing model named MemConFuzz in considering the data flow analysis. This paper makes the following contributions:

(1) A novel algorithm is proposed to obtain locations of heap memory operations by taint analysis based on data dependency. The relation of data dependency is deduced from CPG (Code Property Graph). The location serves as an important indicator for seed selection.
(2) A new algorithm for prioritizing seed selection is proposed based on data dependency for discovering memory consumption vulnerability. Input samples covering more heap operations and data-dependent functions will be assigned high scores, and they are chosen as seeds and assigned more energy in the fuzzing loop.
(3) A novel memory consumption guided fuzzing model, MemConFuzz, is proposed. Compared with AFL [1], MemLock [2], and PerfFuzz [3], MemConFuzz has a significant improvement in discovering memory consumption vulnerability with more quantity and lower time cost.

The rest of the paper is organized as follows. Section 2 introduces related work. Section 3 presents the algorithm for extracting locations of heap operations through taint analysis based on data dependency. In Section 4, the proposed MemConFuzz model is described. In Section 5, the experimental process and the results are discussed. Finally, we conclude the paper in Section 6.

## 2. Related Work

Methods of discovering vulnerability are divided into static techniques and dynamic techniques. Static methods are used to make classification between the target program and known CVE (Common Vulnerabilities and Exposures) code based on structural similarity or statistical similarity by artificial intelligence technology. Dynamic methods include generation fuzzing, coverage-guided fuzzing, and symbolic execution.

Generation fuzzing adopts a generator to create required samples by mapping out all possible fields of the target program. The generator then separately mutates each of these fields to potentially cause crashes. In the generating process, those methods may result in a large number of invalid samples being rejected by the program as they do not follow the correct format. Coverage-guided fuzzing models integrate instrumentation into the target program before tracing the running information. To discover the special target areas in the program, a directed greybox fuzzing is proposed. Symbolic execution analyzes the target program to determine what inputs cause each part of this program to execute. Through symbolic execution, the required samples that execute the constraint code path to reach the target basic block are solved by an SMT (Satisfiability Modulo Theories) solver.

### *2.1. Static Techniques Based on Artificial Intelligence*

During the research of discovering the vulnerability, the bottlenecks are related to how to generate good samples, how to improve path coverage, and how to provide more knowledge support for dynamic methods. Artificial intelligence has been used in the field of vulnerability discovery in recent years.

Machine learning is the most important technology of artificial intelligence, which attains knowledge about features obtained by analyzing an existing vulnerability-related

dataset. This knowledge can be used to analyze new objects and thus predict potentially vulnerable locations in static mode. Machine learning methods can be divided into traditional machine learning, deep learning, and reinforcement learning.

Rajpal [4] used neural networks to learn patterns in past samples to highlight useful locations for future mutations, and then improved the AFL approach. Samplefuzz [5] combined learn and fuzz algorithms to leverage learned samples' probability distribution to make the generation of grammar suitable samples by using past samples and a neural network-based statistical machine learning. NEUZZ [6] leveraged neural networks to model the branching behavior of programs, generating interesting seeds by strategically modifying certain bytes of existing samples to trigger edges that had not yet been executed. Angora [7] modeled the target behavior, treated the mutation problem as a search problem, and applied the search algorithm in machine learning, which used a discrete function to represent the path from the beginning of the program to a specific branch under path constraints, and thus used the gradient descent search algorithm to find a set of inputs that satisfied the constraint and make the program go through that particular branch. Cheng [8] used RNNs to predict new paths of the program and then fed these paths into a Seq2Seq model, increasing the coverage of samples in PDF, PNG, and TFF formats. SySeVR [9] proposed a systematic framework for using deep learning to discover vulnerabilities. Based on Syntax, Semantics, and Vector Representations, SySeVR focuses on obtaining program representations that can accommodate syntax and semantic information pertinent to vulnerabilities. VulDeePecker [10] is a deep learning-based vulnerability detection system, which has presented some preliminary principles for guiding the practice of applying deep learning to vulnerability detection. μVulDeePecker [11] proposed a deep learning-based system for multiclass vulnerability detection. It introduced the concept called code attention to learn local features and pinpoint types of vulnerabilities.

However, most of these works are computationally intensive. The cost is very high because deep learning requires a large amount of data and computing power. The quality and quantity of the training data set have a direct impact on the accuracy of the training model, and there is also a key challenge to accurately locate the instructions where the vulnerability occurs.

### *2.2. Dynamic Execution Fuzzing Technique*

Fuzzing has gained popularity as a useful and scalable approach for discovering software vulnerabilities. In the process of dynamic execution, that is, the fuzzing loop, the fuzzer generally uses the seed selection algorithm to select favorable seeds based on the feedback information of PUT execution, and then performs seed mutation according to a series of strategies to generate new samples and explore paths of the target program. Fuzzing is widely used to test application software, libraries, kernel codes, protocols, etc. Furthermore, symbolic execution is another important approach that can create a sample corresponding to a specific constraint path by the SMT solver. The following mainly introduces several popular dynamic technologies and methods in fuzzing.

A. Coverage-guide fuzzing

Coverage-guide greybox fuzzing (CGF) is one of the most effective techniques to discover vulnerabilities. CGF usually uses path coverage information to guide path exploration. In order to improve the coverage of fuzzers, researchers have focused on optimizing the coverage guide engine, which is the main component of fuzzers.

LibFuzzer [12] provided samples into the library through a specific fuzzing entry point, used LLVM's SanitizerCoverage tool to obtain code coverage, and then performed mutations on the sample to maximize coverage. Honggfuzz [13] proposed a genetic algorithm to efficiently mutate seeds. AFL [1] is a coverage-based fuzzing tool that captures basic block transitions by instrumentation and records the path coverage, thereby adjusting the samples to improve the coverage and increase the probability of finding vulnerabilities. OSS-FUZZ [14] was a common platform built by Google to support fuzzing engines in combination with Sanitizers for fuzzing open source programs. GRIMOIRE [15], Supe-

rion [16], and Zest [17] leveraged the knowledge in highly structured files to generate well-formed samples and traced the coverage of the program to reach deeper levels of code. Therefore, branch coverage was increased. CollAFL [18] proposed a coverage-sensitive fuzzing scheme to reduce path conflicts and thus improve program branch coverage. TensorFuzz [19] used the activation function as the coverage indicator and leveraged the algorithm of fast-approximate nearest neighbor to check whether the coverage increases to accordingly adjust the neural network. PerfFuzz [3] generated input samples by using multi-dimensional feedback and independently maximizing execution counts for all program locations. Fw-fuzz [20] obtained the code coverage of firmware programs of MIPS, ARM, PPC, and other architectures through dynamic instrumentation of physical devices, and finally implemented a coverage-oriented firmware protocol fuzzing method. T-fuzz [21] used coverage to guide the generation of input, and when the new path could not be accessed, the sanity check was removed to ensure that the fuzzer could continue to discover new paths and vulnerabilities.

Most coverage-based fuzzers treat all codes of a program as equals. However, some vulnerabilities hide in the corners of the code. As a result, the efficiency of CGF suffers and efforts are wasted on bug-free areas of the code.

B. Symbolic execution

Symbolic execution is a technique to systematically explore the paths of a program, which executes programs with symbolic inputs. When used in the field of discovering vulnerabilities, symbolic execution can generate new input samples that have a path reaching target codes from the initial code by solving path constraints with the SMT solver. It can also be said to deduce input from results under constraints.

Driller [22] leveraged fuzzing and selective concolic execution in a complementary manner. Angr [23], which is based on the model popularized and refined by S2E [24] and Mayhem [25], was used by Driller to be a dynamic symbolic execution engine for the concolic execution test. Driller uses selective concolic execution to only explore the paths deemed interesting by the fuzzer and to generate inputs for conditions that the fuzzer cannot satisfy. SAGE [26] is equipped with whitebox fuzzing instead of blackbox fuzzing, with symbolic execution to record path information and constraint solvers to explore different paths. QSYM [27] adopted a symbol execution engine for a greybox fuzzing approach to reach deeper code levels of the program. SAFL [28] augmented the AFL fuzzing approach by additionally leveraging KLEE as the symbolic execution engine.

However, the disadvantage of symbolic execution is that the increased analysis process leads to the program running overhead. In addition, as the depth of the path increases, the path conditions will become more and more complex, which will also pose a great challenge to the constraint solver.

C. Directed greybox fuzzing

Directed Greybox Fuzzing (DGF) is a fuzzing approach based on the target location or the specific program behavior obtained from the characteristics of a vulnerable code. Unlike CGF, which blindly increases path coverage, DGF aims to reach a predetermined set of places in the code (potentially vulnerable parts) and spends most of the time budget getting there, without wasting resources emphasizing irrelevant parts.

AFLgo [29] and Hawkeye [30] used distance metrics in their programs to perform user-specified target sites. A disadvantage of the distance-based approach is that it only focuses on the shortest distance, so when there are multiple paths to the same goal, longer paths may be ignored, resulting in lower efficiency. MemFuzz [31] focused on code regions related to memory access, and further guided the fuzzer by memory access information executed by the target program. UAFuzz [32] and UAFL [33] focused on UAF vulnerability-related code regions, leveraging target sequences to find use-after-free vulnerabilities, where memory operations must be performed in a specific order (e.g., allocate, free then store/write). Memlock [2] mainly focused on memory consumption vulnerabilities, took memory usage as the fitness goal, and searched for uncontrolled memory consumption vulnerabilities, but did not consider the influence of data flow. AFL-HR [34] triggered

hard-to-show buffer overflow and integer overflow vulnerabilities through coevolution. IOTFUZZER [35] used a lightweight mechanism based on IoT mobile device APP, and proposed a black-box fuzzing model without protocol specifications to discover memory corruption vulnerabilities of IoT devices.

However, these works focus more on specific measurement strategies. When looking for the optimal path, it is easy to get stuck in local blocks of the program and ignore other paths that may lead to vulnerabilities, thus making the fuzzing results inaccurate.

D. Data flow guided fuzzing

Data flow analysis increases the knowledge set of the fuzzer and semantic information of the PUT by adding data flow information, and thus essentially makes the code characteristics and program behavior clear. Data flow analysis methods, such as taint analysis, can reflect the impact of the mutation on samples that could help optimize seed mutation strategy, input generation, and the seed selection process.

SemFuzz [36] tracked kernel function parameters on which key variables depend through reverse data flow analysis. SeededFuzz [37] proposed a dynamic taint analysis (DTA) approach to identify seed bytes that influence the values of security-sensitive program sites. TIFF [38] proposed a mutation strategy to infer input types through in-memory data structure identification and DTA, which increased the probability of triggering memory corruption vulnerabilities. However, data flow analysis, especially DTA, often increases runtime overhead and slows down the program while obtaining accurate data information of PUT. Fairfuzz [39] and Profuzzer [40] all adopted lightweight taint analysis to find the guiding mutation solution and obtain the variable taint attributes. GREYONE [41] equipped fuzzing with lightweight Fuzzing-Based Taint Inference (FTI) to carry out taint calibration for the branch jump variables of the program control flow. In the process of fuzzing, they mutate the specific bytes of samples and observe the changes of tainted variables to obtain the data dependency relationship between seed bytes and tainted variables.

However, it is impossible to understand the semantics of control flow by simply using data flow for vulnerability discovery, and detailed data flow analysis will increase overhead and reduce fuzzing efficiency. Usually, it can only be used as an important supplementary method of vulnerability discovery based on control flow analysis.

In summary, data flow analysis has become a future research trend, as more additional information of PUT can be obtained for better guidance of fuzzers. Therefore, the performance of the fuzzer can be better played for different vulnerabilities.

## 3. Enhanced Heap Operation Location Based on Data Semantics

In order to focus on discovering heap vulnerability, we first analyze the program in static mode to identify the locations of heap operation. We not only try to obtain the subsequence of heap operation, but also deduct the relations of heap operation based on data semantics. To achieve this goal, we build CPG including CFG and DDG (Data Dependency Graph). CFG is used to describe the sequence of operations, while DDG is used to point out the relationship between heap pointers. Based on data dependency deduced from CPG, we propose an algorithm to extract the locations of suspected dangerous heap operation code areas.

### *3.1. Examples of Memory Consumption Vulnerability*

If an attacker can control the allocation of limited software resources and use a large number of system resources, the attacker may consume all available resources and then trigger a denial of service attack, which belongs to the category of resource consumption vulnerability CWE-400. This kind of vulnerability may prevent authorized users from accessing the software and have harmful effects on the surrounding memory environment. For example, a memory exhaustion attack could render software or even the operating system unusable. Therefore, we focus on the heap memory consumption vulnerability of code blocks, which is divided into two types named uncontrolled memory allocation and memory leaks.

**Definition 1.** ***Memory consumption*** *is defined as a vulnerability occupying process memory resources by triggering data storage instructions several times, which affects the normal running of the process and leads to a denial-of-service attack.*

**Definition 2.** ***Uncontrolled memory allocation*** *is defined as a vulnerability related to heap memory allocation and release, which allocates memory based on untrusted size values, but does not validate or incorrectly validate the size, and allows any amount of memory to be allocated. Its CWE number is CWE-789.*

**Definition 3.** ***Memory leak*** *is defined as a vulnerability also related to heap memory allocation and release, in which the program does not adequately track and free the allocated memory after allocation, and thus slowly consumes the remaining memory. Its CWE number is CWE-401.*

Compared with non-memory consuming vulnerabilities, uncontrolled memory allocation vulnerability and memory leak vulnerability are more difficult to discover because their conditions of triggering crashes are stricter.

CVE-2019-6988 is a public CVE, and this vulnerability occurs in the *opj_calloc* function. This vulnerability is formed because the program code lacks the detection of the allocated amount and the security mechanism for specially crafted files. In Figure 1, the code snippet related to an uncontrolled memory allocation vulnerability (CVE-2019-6988) exists in the executable program OpenJPEG version 2.3.0. In the source code project, the function *opj_tcd_init_tile* in file tcd.c is called when the OpenJPEG is running to decompress the "specially-crafted" images. This vulnerability allows a remote attacker to attempt too much memory allocation by function *opj_calloc* in the file opj_malloc.c, which calls the system function *calloc* to allocate a large amount of heap memory and ultimately results in a denial-of-service attack due to a lack of enough free heap memory.

```
1  static INLINE OPJ_BOOL opj_tcd_init_tile(opj_tcd_t *p_tcd, OPJ_UINT32 p_tile_no,
2          OPJ_BOOL isEncoder, OPJ_FLOAT32 fraction, OPJ_SIZE_T sizeof_block,
3          opj_event_mgr_t* manager){
4      ...
5      // call opj_tgt_create to creat the file.
6      if (! l_current_precinct->incltree) {
7          l_current_precinct->incltree = opj_tgt_create(l_current_precinct->cw,
8                                                        l_current_precinct->ch, manager);
9      } else {
10         l_current_precinct->incltree = opj_tgt_init(l_current_precinct->incltree,
11                                                     l_current_precinct->cw, l_current_precinct->ch, manager);
12     }
13     ...
14 }
15
16 opj_tgt_tree_t *opj_tgt_create(OPJ_UINT32 numleafsh, OPJ_UINT32 numleafsv,
17                                opj_event_mgr_t *p_manager){
18     ...
19     tree = (opj_tgt_tree_t *) opj_calloc(1, sizeof(opj_tgt_tree_t)); // call opj_calloc.
20     if (!tree) {
21         opj_event_msg(p_manager, EVT_ERROR, "Not enough memory to create Tag-tree\n");
22         return 00;
23     }
24     ...
25 }
26 void * opj_calloc(size_t num, size_t size){
27     if (num == 0 || size == 0) {
28         return NULL;
29     }
30     return calloc(num, size); // call calloc.
31 }
```

**Figure 1.** Code snippet from tcd.c/tgt.c in OpenJPEG v2.3.0.

As shown in Figure 2, the code snippet concerning memory leaks vulnerability exists in a program case of Samate Juliet Test Suite. This case is a memory leak vulnerability caused by allocating heap memory without release. Specifically, the case uses the function *malloc* on line 5 to allocate memory and checks whether the allocation is successful or not

on line 7. However, at the end of the function, the allocated memory data is not effectively released, eventually resulting in a heap memory leak.

```
1  void func()
2  {
3      int64_t * data;
4      data = NULL;
5      data = (int64_t *)malloc(100*sizeof(int64_t));
6
7      if (data == NULL){
8          exit(-1);
9          }
10     data[0] = 5LL;
11     printLongLongLine(data[0]);
12 }
```

**Figure 2.** Code snippet from Samate Juliet Test Suite.

### *3.2. Location of Heap Operation Code Based on Data Semantic*

In order to directionally discover heap-memory-consumed vulnerabilities, how to obtain the locations of suspected heap operations is the first essential goal. Once the locations are identified, they will be used as a guided factor to optimize the guidance strategy of vulnerability fuzzing, which is our second essential goal.

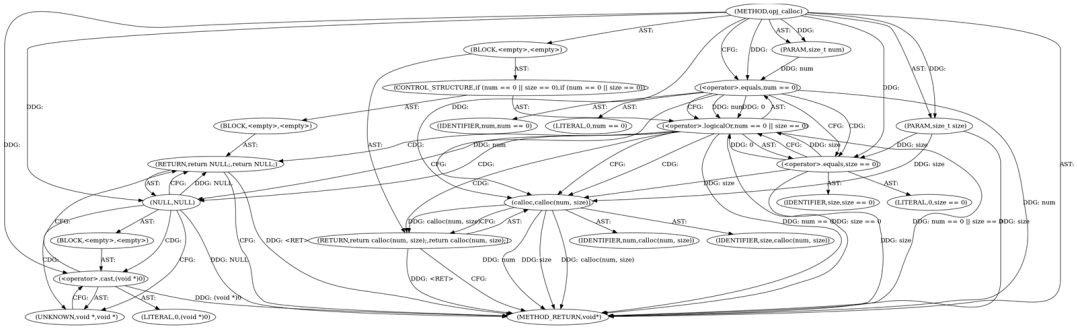
We first construct CPG based on the static analysis tool Joern. Then, a scheme is proposed to deduce the explicit and implicit semantic relations between heap pointers based on data flows from CPG. In addition, based on the semantic relations between heap pointers, we analyze the abnormal sequence of heap memory operation concerning allocation and release, and thus demarcate the heap operation code areas with suspected heap consumption. These locations will serve as an important indicator for selecting seeds from input samples during the fuzzing procedure.

#### 3.2.1. Construct CPG

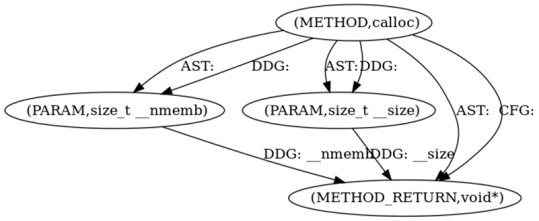
CPG is a graph combining multi-level code information where the information at each level can be related to each other. CPG can be obtained by combining AST (Abstract Syntax Trees), CFG, DDG, and CDG (Control Dependency Graph). Compared with other structures, CPG contains much richer data and relational information, which enables more complex and detailed static analysis of the program source code.

The CPG is composed of nodes and edges. Nodes represent the components of PUT, including functions, variables, etc. Each node has a type, such as a type METHOD representing a method, PARAM representing a parameter, and LOCAL representing a local variable. The directed edges represent the relationship between nodes, and the label is the description of the relationship, such as a label DDG from node A to node B represents B's data dependency on A.

The program files can be parsed using the source code analysis tool Joern to obtain the CPG. In order to show what useful data can be obtained from CPG for data relationship derivation, we analyze OpenJPEG v2.3.0 containing CVE-2019-6988 introduced in Section 3.1. Due to the huge number of codes, we only show the partial CPG shown in Figure 3. Figure 3a is the full CPG of the *opj_calloc* function, in which the *calloc* method is the partial zoom shown as Figure 3b. From Figure 3b, we can find the *calloc* method is dependent on the parament t_nmemb and t_size. We also find the parament t_nmemb and t_size are dependent on the return method. Combined with CFG, we can derive the potential dependent relationship between the *calloc* function and the *calloc* function and the return function.




(**a**) **CPG of *opj_calloc* method.**




(**b**) **CPG of *calloc* method.**

**Figure 3.** CPG of OpenJPEG v2.3.0 *opj_calloc* and *calloc* methods.

Therefore, after constructing the CPG of the program, we analyze data dependencies using taint analysis on CPG and determine the location of heap operations.

#### 3.2.2. Location Extraction Based on Data Dependency

Current research faces the challenge of finding accurate locations for code areas related to heap operations. In this section, we introduce how to obtain the data dependency by taint analysis. Taint analysis is an effective technology for data flow analysis. In our research work, we use a lightweight static taint analysis method to locate potentially vulnerable code areas.

Because CFG can reflect the jump of code and show all branches, most state-of-the-art fuzzers use CFG as an analysis object. Meanwhile, the data flow can reflect the direct relationship between variables and the function parameters, so some fuzzers also consider data flow as the analysis object. The data flow and the dependence on data semantics can provide positive help for understanding the real behavior of CFG, so we use these advantages to better serve the seed selection for discovering our required types of vulnerabilities. Using CPG for program analysis has many advantages. After using Joern to parse the source code into CPG, it does not need to be further compiled. CPG will be loaded into memory, and we can perform traversal queries, evaluate function leakage problems, perform data flow analysis, etc.

Dynamic taint analysis usually increases program runtime overhead. To this end, we use a lightweight static data flow analysis method to obtain the suspected locations in the target program, thereby reducing the impact on the program runtime overhead. We mainly focus on data dependencies in CPG during static taint analysis. We analyze the CPG to obtain relevant function points that have data dependencies between program input and memory allocation, that is, taint attributes information, and record them. Specifically, we use a static taint analysis approach to obtain the location of functions including heap operation.

Algorithm 1 is proposed to extract location by taint analysis based on data dependency. The static taint analysis is used to track the data flow of heap operation functions such as *malloc*, *calloc*, *realloc*, *free*, *new*, *delete*, and their deformation functions. The source set of the algorithm is the parameter of all the program methods and all the called functions of the program, and the sink set is the function arguments. In the process of data flow analysis, all relevant nodes from source to sink are traversed, and we use Joern's built-in functions, such

as *reachableBy* and *reachablebyFlows*, to query the paths from all sources to the sink points in its CPG. Finally, matched functions are obtained, and duplicated ones are removed.

**Algorithm 1:** Taint analysis approach for locating potentially vulnerable functions

**Input:** *CPG* of program under test *P*
**Output:** Set of dataflow functions $Set_{funcs}$

$Set_{funcs} = \emptyset$;
Target heap functions *funcs* ← {*malloc, calloc, free...*};
**for** *f in funcs* **do**
  *Source* = called functions, methods' *params* of *P* in *CPG*;
  *Sink* = *args* of *f* in *CPG*;
  **if** *Sink dataFlowReachable by Source* **then**
    $Nodes_{related} = \emptyset$; //nodes which are data related;
    $Nodes_{related} \leftarrow Nodes_{related} \cup$ Traverse(CPG of P);
    *paths* = Query($Nodes_{related}$, *CPG*); //dataflow paths of heap opreations;
  **for** *p in paths* **do**
    (*statements, line num,* $funcs_p$*, source locations*) ← regularMatch(*p*);
    **if** $funcs_p$ *not* $\emptyset$ **then**
      Remove duplicate items in $funcs_p$;
      $Set_{funcs} \leftarrow Set_{funcs} \cup funcs_p$;
**return** $Set_{funcs}$;

Figure 4 shows the partial data flow of the heap memory allocation function obtained through static taint analysis in OpenJPEG v2.3.0, which is a data flow path to the parameter value of the standard library function *malloc*. Each path contains four aspects of information. Among them, the tracked column contains the statements in the queried nodes, the lineNumber column contains the line number in the source code file, the method column displays the method names where the statements are located, and the file column displays the locations of the source code file. To construct the source set, we mark the parameters of all methods and call all functions in the CPG into the source set. We find all call-sites for all methods in the graph with the name *malloc* and mark their arguments into the sink set. After identification, we obtain a data flow path to *malloc* in our query of OpenJPEG's CPG. Eventually, we collect dataflow-related functions *jpip_to_jp2*, *fread_jpip*, and *opj_malloc*, which were obtained in the dataflow path after the static taint analysis.

| tracked | lineNumber | method | file |
|---|---|---|---|
| jpip_to_jp2(char *argv[]) | 45 | jpip_to_jp2 | /src/bin/jpip/opj_jpip_transcode.c |
| fread_jpip(argv[1], dec) | 51 | jpip_to_jp2 | /src/bin/jpip/opj_jpip_transcode.c |
| fread_jpip(const char fname[], jpip_dec_param_t *dec) | 350 | fread_jpip | /src/lib/openjpip/openjpip.c |
| open(fname, O_RDONLY) | 354 | fread_jpip | /src/lib/openjpip/openjpip.c |
| open(fname, O_RDONLY) | 354 | fread_jpip | /src/lib/openjpip/openjpip.c |
| infd = open(fname, O_RDONLY) | 354 | fread_jpip | /src/lib/openjpip/openjpip.c |
| get_filesize(infd) | 359 | fread_jpip | /src/lib/openjpip/openjpip.c |
| get_filesize(infd) | 359 | fread_jpip | /src/lib/openjpip/openjpip.c |
| (Byte8_t)get_filesize(infd) | 359 | fread_jpip | /src/lib/openjpip/openjpip.c |
| dec->jpiplen = (Byte8_t)get_filesize(infd) | 359 | fread_jpip | /src/lib/openjpip/openjpip.c |
| opj_malloc(dec->jpiplen) | 363 | fread_jpip | /src/lib/openjpip/openjpip.c |
| opj_malloc(size_t size) | 191 | opj_malloc | /src/lib/openjp2/opj_malloc.c |
| size == 0U | 193 | opj_malloc | /src/lib/openjp2/opj_malloc.c |
| malloc(size) | 196 | opj_malloc | /src/lib/openjp2/opj_malloc.c |

**Figure 4.** OpenJPEG v2.3.0 partial data flow path.

In summary, the proposed algorithm 1 analyzes the data flow related to heap memory allocation and release in the program, and obtains the locations, variables, and parameters related to heap operation, which guide seed selection in the following fuzzing process.

## 4. MemConFuzz Model

After analyzing the CPG in the static analysis stage, we obtain the function locations related to the data flow of the heap memory allocation and release functions and quantitatively record the sizes of the memory block allocated and released by the heap operations. We feed these back to the fuzzer to prioritize the detection of relevant vulnerable code areas in the fuzzing loop.

The prioritizing discovery of consumption-type vulnerabilities is a novel contribution to this paper. Through the calibration of suspected heap memory-consuming instructions, the priority discovering of them is realized. Through investigating the existing vulnerability discovery models, we found that there are few studies on the discovery of heap memory consumption vulnerabilities. At the same time, the existing methods for discovering such vulnerabilities have many deficiencies, such as the lack of the important data flow analysis related to heap operations. Therefore, we propose our model, which has benefits for our purpose of focusing on heap consumption vulnerabilities discovery, while taking into account the discovery of other vulnerabilities.

### *4.1. Overview*

To address problems mentioned in the previous sections, we propose a memory consumption-guided fuzzing model, MemConFuzz, as shown in Figure 5. The main components of MemConFuzz contain a static analyzer, an executor, fuzz loop feedback, a seed selector, and a seed mutator. In MemConFuzz, the static analyzer marks the dataflow-related edges and records the trigger value for each edge by scanning the source code and then inserts code fragments to update the value in the running program. The executor executes the instrumented program. Fuzz loop feedback is used to record and update related information to guide the seed selector after the program execution. The seed selector adopts a priority strategy to select seeds according to the different scores of the seed bank. The seed mutator mutates the selected seed to test the program in the fuzzing loop.

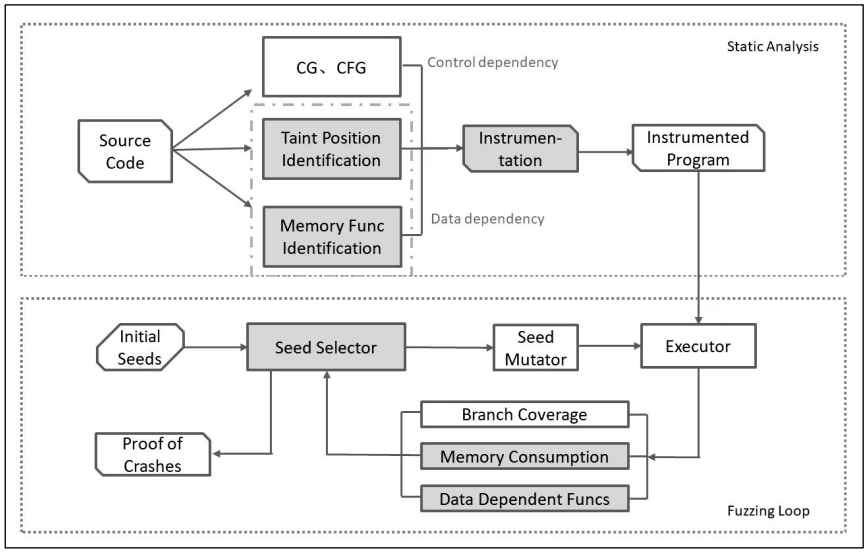



**Figure 5.** MemConFuzz model.

MemConFuzz contains two main stages: the static analysis stage and the fuzzing loop stage. Dark colors indicate optimized changes to the original AFL approach. Static analysis performs taint position identification and memory function identification for instrumentation. We use lightweight instrumentation to capture basic block transitions, heap memory function locations, and data-flow-dependent function locations at compile-time, while gaining coverage information, heap memory size, and data-flow-dependent information during runtime.

In the static analysis stage, we instrument all the captured target locations and then recompile to obtain an instrumented file. In the main fuzzing loop, the seed is selected for mutation and delivered to the instrumented file for execution. The model continuously tracks the state of the target program and records the cases that cause the program to crash. At the same time, the recorded feedback information is continuously submitted to the seed selector for priority selection, which helps to discover more heap memory consumption vulnerabilities.

### *4.2. Code Instrumentation at Locations of Suspected Heap Operation*

In order to record the execution information in the fuzzing process, the bitmap of AFL records the number of branch executions, and the *perf_bits* of Memlock records the size of the heap allocation. The MemConFuzz also adopts a shared memory and incrementally adds *dataflow_shm* to store the numbers of the data-dependent functions triggered.

The MemConFuzz is derived from AFL. MemConFuzz adds two shared memory areas in AFL and mainly expands the afl-llvm-rt.o.c and afl-llvm-pass.so.cc files for instrumentation. The instrumented contents include branch coverage information, heap memory allocation functions, and data-dependent functions.

The first shared memory *perf_bits* records the size of the memory allocation and release during runtime. In the static analysis stage, we use LLVM [42] to obtain the function Call Graph (CG) and CFG of the program. Through traversing CG and CFG, we search the locations of basic blocks related to heap memory allocation and release functions, including *malloc*, *calloc*, *realloc*, *free*, *new*, *delete*, and their variant functions, and locate the call-sites of heap functions for instrumentation. During the fuzzing loop, *perf_bits* records the amount of consumed heap memory.

As shown in Figure 6 below, there are four basic blocks A, B, C and D representing nodes in the CFG of the program. The program will first go to B or C according to a branch condition. Once the branch condition for block C is met, the variable *size* is initialized, and then the memory allocation operation is performed in the block D. We traverse branches of the basic block described by IR language from the beginning of the program. Once we find a match among all our target heap functions, we locate the potential block D and instrument it.

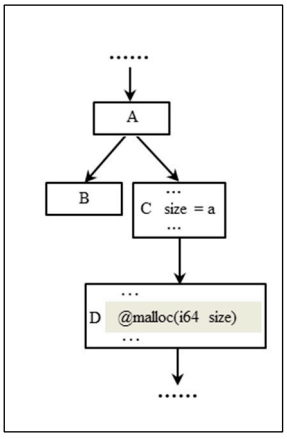



**Figure 6.** An example of basic block transition.

Meanwhile, we add the second shared memory *dataflow_shm* to record the numbers of data-dependent functions. We traverse the basic blocks of the program to search the locations that belong to $Set_{funcs}$, and then complete instrumentation. Specifically, after using these locations to analyze the program, we instrument the code of increasing count in *dataflow_shm*. In the fuzzing loop stage, MemConFuzz can increase the count value in *dataflow_shm* corresponding to triggered data-dependent functions when the target program executes an input sample. Thus, we can get the coverage information of heap operation when the execution of an input sample is completed.

In the instrumentation pass file, we declare a pointer variable, *DataflowPtr*, pointing to the shared memory area *dataflow_shm*. Then, the values of *dataflow_shm* are changed based on the number of data-dependent functions triggered. We inject instrumentation codes into the program during compilation. The approximate formulas for instrumentation are shown below, where Formula (1) marks the ID of the current block with a random number *cur_location*, Formula (2) shows that by applying the XOR operation on two IDs of the current block and previous block as the key, the corresponding value in *dataflow_shm* is updated by adding *dataflowfunc_cnt* to self-value, where the *shared_mem[]* array is our *dataflow_shm*, the size of which is 64 Kb, and *dataflowfunc_cnt* is the count of data-dependent functions triggered on this branch, and, in Formula (3), in order to distinguish the paths in different directions between two blocks, the *cur_location* is moved to the right by one bit as the *prev_location* to complete the marking of these two blocks.

$$cur\_location = \text{<COMPILE\_TIME\_RANDOM>} \tag{1}$$

$$shared\_mem[cur\_location \hat{} prev\_location] += dataflowfunc\_cnt \tag{2}$$

$$prev\_location = cur\_location >> 1 \tag{3}$$

We instrument the program based on static analysis to get the instrumented program. Therefore, we prioritize guidance to the suspected heap operation areas in the fuzzing loop stage to realize our directed fuzzing on the heap consumption vulnerability.

### *4.3. Strategy of Seed Priority Selection*

This model proposes a fine-grained seed priority strategy for discovering heap memory consumption vulnerabilities. Seeds are mainly scored by the following indicators:

(1) We use *dataflow_funcs* as the metric, which is instrumented during the static analysis stage to record the number of data-dependent functions triggered during execution. The more related functions that are triggered, the higher the seed priority.
(2) Like Memlock, we record the size of the allocated heap memory; the larger the heap memory that is allocated, the more power this input sample gets. The input samples with more power at the top of the queue are selected as seeds. We use *new_max_size* as a flag which represents the maximum memory newly consumed on the heap in history. When the flag is triggered, we increase the score of this seed and enlarge its mutation time.
(3) In addition, when no data-dependent function has triggered and the new maximum allocated memory has not been reached, MemConFuzz still retains AFL's path-coverage-based seed prioritization strategy to cover as many program branches as possible.

The first two strategies will help the fuzzer trigger more potential heap memory consumption vulnerabilities.

**Principle 1.** *During execution, the more data dependent functions that are recorded, the greater the coefficient increase. In the end, the seed score increases and the energy obtained increases.*

**Principle 2.** *The original scoring strategy of AFL should also be taken into account. The final score of the seed should not be too large, because the execution process may be trapped in local code blocks of the program.*

According to the above two seed selection formula design principles, and in order to evaluate the excellence degree of each input sample, a scoring formula is proposed, in which the more data-dependent functions are recorded, the larger the coefficients are increased, and the higher scores are achieved. In addition, we set the parameters 1.2 and 1/5 according to the design principle, making sure to set the multiplier factor of the increase to the maximum value of 1.2, so that the evaluating strategy does not have too much impact

to avoid missing out on other good samples, which are not used for discovering memory consumption vulnerability, but can be used for non-memory consumption vulnerability.

$$Priority\_score(sample_i) = \begin{cases} P_{afl}(sample_i) \cdot \left(1.2 - \frac{1}{5} e^{-dataflow\_funcs}\right), & dataflow\_funcs \;||\; new\_max\_size \\ P_{afl}(sample_i) & , \quad otherwise \end{cases} \tag{4}$$

Equation (4) shows the seed priority strategy adopted by MemConFuzz. Specifically, for each $sample_i$ in the sample queue, when the data-dependent function is triggered or the new maximum memory is reached, we multiply the original AFL score value $P_{afl}(sample_i)$ by our formula and then obtain different seed scores under different numbers of data-dependent functions. The *dataflow_funcs* is the total number of data-dependent functions triggered by the sample during the fuzzing loop. Otherwise, we adopt the original AFL strategy, which is to perform sample scoring according to the execution speed and length of the samples. At last, we choose some samples with high *Priority_score* values from the sample queue as seeds.

In summary, every time the program executes, we detect code coverage, memory usage, and data-dependent functions triggered. For the impact of heap operations, we adopt two equations for different cases. The samples that trigger more data-related functions, allocate larger heap memory sizes, and have higher program path coverage are preferentially given higher power. Furthermore, we set up a maximum time in the havoc mutation phase to prevent wasting too much test time.

### *4.4. Proposed Model*

We implement a directed fuzzing model MemConFuzz to discover heap memory consumption vulnerabilities. Unlike AFL, our model first performs a static analysis approach to analyze program data flow, and then uses the data flow information as a guide in discovering heap memory consumption vulnerabilities. Algorithm 2 describes the workflow of MemConFuzz.

The current vulnerability discovery models are faced with the challenge of not being able to prioritize the discovery of heap memory consumption vulnerabilities, and there is a problem of inaccurate static analysis caused by a lack of data flow information. Algorithm 2 is the pseudo code of our proposed fuzzing model, which is improved based on AFL, and we have optimized and improved the seed selection.

In the seed queue *Queue*, we select a seed $q$ based on our seed priority strategy and then assign energy to the mutation. Meanwhile, we record the hashes, memory size, and data-dependent functions in each running process. If the $q'$ causes the program to crash, add it to the crash set. Otherwise, we select those seeds that can trigger a new path, more heap memory allocation, or trigger more data-dependent functions, set them as interesting samples, and add them to the seed queue for the mutation of the next loop. Finally, the collection set of seeds that trigger heap memory consumption vulnerabilities and cause crashes is obtained.

**Algorithm 2:** Memory Consumption Fuzzing

**Input:** Instrumented program $P$, Initial seed input $S$
**Output:** Set of crash outputs $Set_{crash}$

$Set_{crash} = \emptyset$;
$Queue \leftarrow S$;
**while** *time and resource budget do not expire* **do**
  **if** $Queue \not\emptyset$ then
    $q$ = ChooseNext($Queue$); // Our Modifications;
    $e$ = AssignEnergy($q$);
    **if** *i from 1 to e* **then**
      $q'$ = Mutate($q$);
      ($tracebits_i$, $memory_i$, $dataflowfuncs_i$) $\leftarrow$ Run($q'$, $P$);
      $hash_i$ = Hash($tracebits_i$);
      **if** $q'$ *triggers crash* **then**
        $Set_{crash} \leftarrow Set_{crash} \cup q'$;
      **else**
        **if** *NewCoverage*($q'$) **then**
          $Queue \leftarrow Queue \cup q'$;
        **if** *NewMaxSize*($q'$) **then**
          $Queue \leftarrow$ Update($q'$, $memory_i[hash_i]$);
        **if** *DataflowFuncs*($q'$) **then**
          $Queue \leftarrow$ Add and Prioritize($q'$, $dataflowfuncs_i$);
**return** $Set_{crash}$;

## 5. Experimental Results and Discussions

We implement the MemConFuzz based on the AFL-2.52b framework. We mainly write additional codes for LLVM-mode (based on LLVM v6.0.0) to realize our program static analysis approach related to memory consumption based on data flow and modify *afl-fuzz.c* to support our interaction module with instrumentation information and the fine-grained seed priority selection strategy.

We chose popular open source programs OpenJPEG v2.3.0, jasper v2.0.14, and readelf v2.28 with heap memory consumption vulnerabilities as test datasets, and compared them against AFL, MemLock, and PerfFuzz. Our experiments were performed on Ubuntu LTS 18.04 with a Linux kernel v4.15.0, Intel(R) Xeon(R) CPU E7-4820 processor, and 4GB RAM. The experiment results show that MemConFuzz outperforms the state-of-the-art fuzzing techniques, including AFL, MemLock, and PerfFuzz, in discovering heap memory consumption vulnerabilities. MemConFuzz can discover heap memory consumption CVEs faster and trigger a higher number of heap memory consumption crashes.

### *5.1. Evaluation Scheme*

During the experiment, since the fuzzer heavily relies on random mutations, there may be performance fluctuations between different experiments on our machine, resulting in different experimental results each time. We have taken effective measures to configure experimental parameters and have taken two measures to mitigate the randomness caused by the properties of the fuzzing technology. First, we conduct a uniform long-term test of the experimental process of each PUT performed by each fuzzer until the fuzzer reaches a relatively stable state. Specifically, our stable results are obtained after a uniform 24-h period during every fuzzing execution. Second, we add the -d option to all fuzzers in the experiment to skip the deterministic mutation stage, so that more mutation strategies can

be performed in the havoc and splicing stages to discover heap memory consumption vulnerabilities.

Due to factors such as different computer performance and randomness of mutation, the results of each experiment will be different. For the experiments in the comparison model, such as Memlock, we reproduced them on the same machine in order to ensure that each model is based on the same initial experimental conditions. We give the definition of "relatively stable state".

**Definition 4.** ***Relatively Stable State*** *is defined as a state in which test data smoothly changes. On the same machine, after a certain period of time, the results of multiple experiments are relatively stable compared to the growth rate in the initial stage, and then the test results reach a "relatively stable state".*

Figure 7 below is an experimental record of fuzzing readelf; the ordinate shows the number, and the abscissa shows the time. We mainly focus on the changes in the number of unique crashes. It shows that the growth rate is the fastest in the first 2 h, and the growth rate slows down after about 22 h, which fully meets the definition of a "relatively stable state". The other tests also meet the definition of a "relatively stable state" around 24 h. We consulted a large number of vulnerability discovery studies and methods, and many studies also selected 24 h as the test standard. In addition, MemConFuzz, Memlock, and PerfFuzz are all improved based on AFL, so if the time is too long, when almost no heap consumption vulnerabilities can be found in the end, it will gradually degenerate into AFL's general vulnerability discovering, and the discovery efficiency for heap consumption vulnerability cannot be demonstrated at this time. In order to comprehensively ensure accuracy and efficiency, we uniformly select 24 h as our test standard, which can reflect the ability of vulnerability discovering and also reduce unnecessary time overhead.

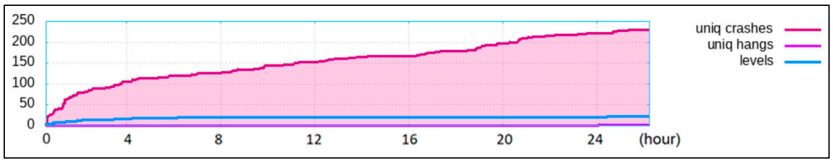



**Figure 7.** Experimental record of fuzzing readelf.

We enable ASAN [43] compilation of the source program file, and set the *allocator_may_return_null* option so that the program will crash when the heap memory allocation fails due to the allocation of too much memory, which is convenient for us to observe and analyze. In addition, we used LeakSanitizer to detect memory leak vulnerabilities and conduct subsequent analyses.

### *5.2. Experimental Results and Discussions*

We perform fuzzing on the selected real-world program datasets and record the experimental data according to the evaluation metrics.

To demonstrate our work, we compare against some fuzzing techniques, recording the number of triggering heap memory consumption vulnerabilities and the time of triggering real-world CVEs. We select large-scale programs with tens of thousands of lines, which are continuously maintained in the open source community and have high popularity. These programs are from the comparison model. The name and version of the test software are mentioned by Memlock and some other fuzz testing tools. They contain heap consumption vulnerabilities and other types of vulnerabilities as interference items to comprehensively evaluate the models. Because we use the analysis method of source code and semantic heap operation code, other corresponding open-source source codes are difficult to find. There are very few fuzzing research works related to this type of vulnerability. In order to better evaluate the horizontal performance of the model, we choose these programs and

ensure that these softwares are publicly available for download. The download link has been added. Additionally, the source code of MemConFuzz will be available for request.

Table 1 shows the crashes related to memory consumption vulnerabilities obtained by fuzzing the programs jasper, readelf, and openjpeg. UA stands for uncontrolled-memory-allocation vulnerabilities, ML stands for memory leak vulnerabilities, and SLoC stands for Source Lines of Code. For each 24-h fuzzing experiment, we use Python to analyze the obtained crashes and automatically reproduce them. We classify the crashes according to the obtained Address Sanitizer function call chain and its output summary information of vulnerability types, and then obtain the memory consumption-related vulnerabilities we need, that is, the number of UA and ML. Among them, most of the crashes triggered by jasper are ML, while the crashes triggered by other programs are UA. The results show that MemConFuzz has an improvement of 43.4%, 13.3%, and 561.2% in the discovery of heap memory consumption vulnerabilities compared with the advanced fuzzing techniques AFL, MemLock, and PerfFuzz, respectively.

**Table 1.** Number of heap memory consumption vulnerabilities.

| Program | Version | SLoC | Type | MemConFuzz | AFL | MemLock | PerfFuzz |
|---|---|---|---|---|---|---|---|
| jasper | 2.0.14 | 44k | UA | **5** | 1 | 2 | 0 |
| | | | ML | 208 | **212** | 190 | 28 |
| readelf | 2.28 | 1844k | UA | **219** | 86 | 182 | 39 |
| openjpeg | 2.3.0 | 243k | UA | 11 | 10 | **17** | 0 |
| Total Unique Crashes (Improvement) | | | | **443** | 309 (+43.4%) | 391 (+13.3%) | 67 (+561.2%) |

The test programs [44–46] selected are all historical versions. After our automated crash analysis, the discovered vulnerabilities are all historically reported vulnerabilities. Our experimental comparison mainly focuses on the number and speed of discovering heap memory consumption vulnerabilities. We may consider discovering and analyzing additional new vulnerabilities in future research.

The AFL framework shows that vulnerabilities with the same crash point belong to the same vulnerability. Vulnerabilities are divided into many types. Since we are targeting heap consumption vulnerabilities, the only thing we need to confirm is whether the discovered vulnerabilities belong to heap consumption vulnerabilities. We wrote automated crash analysis scripts and compared the crash function stacks reported by ASAN. Through the ASAN report, the function call relationship, and the location of the crashed code, we spent a lot of time confirming that the vulnerability mentioned in this experiment belonged to the heap consumption vulnerability.

Furthermore, we also recorded the time of triggering real-world CVEs. In order to facilitate experimental comparison, we conducted a 24-h test for each test, and T/O stands for a timeout during the 24-h test. Table 2 shows the time of real-world CVEs triggered after we fuzzed on our dataset. Likewise, we used ASAN to reproduce crashes to detect memory error information. We did not use Valgrind because it slows the program down too much, while ASAN only slows the program down about 2×. We use Python to automatically analyze crashes and search the crash points, and compare the obtained Address Sanitizer function the call chain and crash point with the function location described by the real-world CVE information, therefore gaining the time of the first matching crash. Our experimental results show that MemConFuzz has significant time reduction compared to the state-of-the-art fuzzing techniques AFL, MemLock, and PerfFuzz, respectively. Among them, CVE-2017-12982 has more obvious advantages, which can make the program allocate large heap memory faster and trigger the vulnerability faster. The reason is that the proposed model focuses on the location of functions that are data-dependent on memory consumption, and pays attention to the size of allocated memory, which is more targeted for memory consumption vulnerabilities than other fuzzing models.

**Table 2.** Trigger time of real-world vulnerability.

| Program | Vulnerability | Type | MemConFuzz | AFL | MemLock | PerfFuzz |
|---|---|---|---|---|---|---|
| | | | Time (h) | Time (h) | Time (h) | Time (h) |
| jasper | CVE-2016-8886 | UA | 2.6 | 10.2 | **1.5** | T/O |
| readelf | CVE-2017-9039 | UA | 0.1 | 0.1 | 0.1 | 0.1 |
| openjpeg | CVE-2017-12982 | UA | **2.2** | 12.8 | 5.5 | T/O |
| | CVE-2019-6988 | UA | **12.5** | T/O | 14.8 | T/O |
| Average Time Usage (Improvement) | | | **4.35** | 11.78 (2.71×) | 5.48 (1.26×) | 18.03 (4.14×) |

## 6. Conclusions and Future Work

In this paper, we propose a directed fuzzing approach MemConFuzz model based on data flow analysis of heap operations to discover heap memory consumption vulnerabilities. The MemConFuzz uses the coverage information, memory consumption information, and data dependency information to guide the fuzzing process. The coverage information guides the fuzzer to explore different program paths, the memory consumption information guides the fuzzer to search for program paths that show increasing memory consumption, and the data information guides the fuzzer to explore paths with increasing dependencies on heap memory data flow. Experimental results show that the MemConFuzz outperforms the state-of-the-art fuzzing technologies, AFL, MemLock, and PerfFuzz, in both the number of heap memory vulnerabilities and the time to discovery.

In the future, we plan to enhance the heap memory consumption vulnerability discovery capabilities and vulnerability coverage of our approach with more efficient and more complete data flow analysis. Furthermore, we will add support for binaries to our proposed vulnerability discovery methodology. We will disassemble the binary code to obtain the instruction code set, complete the analysis of the control flow and the data flow, and discover the heap memory consumption vulnerabilities of the binary program more effectively.

**Author Contributions:** Conceptualization, C.D. and Y.G.; methodology, C.D. and Y.G.; software, Z.C. and G.X.; validation, C.D. and Y.G.; investigation, Z.C. and Z.W.; writing—original draft preparation, Z.C. and Y.G; writing—review and editing, C.D. and Y.G.; visualization, Z.C.; project administration, C.D.; funding acquisition, Z.W. All authors have read and agreed to the published version of the manuscript.

**Funding:** This research was funded by the National Natural Science Foundation of China grant number 62172006 and the National Key Research and Development Plan of China grant number 2019YFA0706404.

**Institutional Review Board Statement:** Not applicable.

**Informed Consent Statement:** Not applicable.

**Data Availability Statement:** The test results data presented in this study are available on request. The data set can be found in public web sites.

**Conflicts of Interest:** The authors declare no conflict of interest.

## Abbreviations

AFL America Fuzzy Lop
PUT Program Under Test
UAF Use-After-Free
CPG Code Property Graph
CVE Common Vulnerabilities and Exposures
SMT Satisfiability Modulo Theories

| | |
|---|---|
| CGF | Coverage-guide Greybox Fuzzing |
| DGF | Directed Greybox Fuzzing |
| DTA | Dynamic Taint Analysis |
| FTI | Fuzzing-Based Taint Inference |
| DDG | Data Dependency Graph |
| AST | Abstract Syntax Trees |
| CDG | Control Dependency Graph |
| CG | Call Graph |
| CFG | Control Flow Graph |

## References


1. Zalewski, M. American Fuzzing Lop. Available online: https://lcamtuf.coredump.cx/afl/ (accessed on 31 October 2022).
2. Wen, C.; Wang, H.; Li, Y.; Qin, S.; Liu, Y.; Xu, Z.; Chen, H.; Xie, X.; Pu, G.; Liu, T. Memlock: Memory usage guided fuzzing. In Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, Seoul, Republic of Korea, 27 June 2020–19 July 2020; pp. 765–777. [CrossRef]
3. Lemieux, C.; Padhye, R.; Sen, K.; Song, D. Perffuzz: Automatically generating pathological inputs. In Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, Amsterdam, The Netherlands, 16–21 July 2018; pp. 254–265.
4. Rajpal, M.; Blum, W.; Singh, R. Not all bytes are equal: Neural byte sieve for fuzzing. *arXiv* **2017**, arXiv:1711.04596.
5. Godefroid, P.; Peleg, H.; Singh, R. Learn&fuzz: Machine learning for input fuzzing. In Proceedings of the 2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE), Urbana, IL, USA, 30 October–3 November 2017; pp. 50–59.
6. She, D.; Pei, K.; Epstein, D.; Yang, J.; Ray, B.; Jana, S. Neuzz: Efficient fuzzing with neural program smoothing. In Proceedings of the 2019 IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 19–23 May 2019; pp. 803–817.
7. Chen, P.; Chen, H. Angora: Efficient fuzzing by principled search. In Proceedings of the 2018 IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 20–24 May 2018; pp. 711–725.
8. Cheng, L.; Zhang, Y.; Zhang, Y.; Wu, C.; Li, Z.; Fu, Y.; Li, H. Optimizing seed inputs in fuzzing with machine learning. In Proceedings of the 2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion), Montreal, QC, Canada, 25–31 May 2019; pp. 244–245.
9. Li, Z.; Zou, D.; Xu, S.; Jin, H.; Zhu, Y.; Chen, Z. SySeVR: A Framework for Using Deep Learning to Detect Software Vulnerabilities. *IEEE Trans. Dependable Secur. Comput.* **2022**, *19*, 2244–2258. [CrossRef]
10. Li, Z.; Zou, D.; Xu, S.; Ou, X.; Jin, H.; Wang, S.; Deng, Z.; Zhong, Y. Vuldeepecker: A deep learning-based system for vulnerability detection. *arXiv* **2018**, arXiv:1801.01681.
11. Zou, D.; Wang, S.; Xu, S.; Li, Z.; Jin, H. µVulDeePecker: A Deep Learning-Based System for Multiclass Vulnerability Detection. *IEEE Trans. Dependable Secur. Comput.* **2021**, *18*, 2224–2236. [CrossRef]
12. LibFuzzer—A Library for Coverage-Guided Fuzz Testing. Available online: http://llvm.org/docs/LibFuzzer.html (accessed on 31 October 2022).
13. Honggfuzz. Available online: http://honggfuzz.com/ (accessed on 31 October 2022).
14. Serebryany, K. OSS-Fuzz-Google's Continuous Fuzzing Service for Open Source Software. In Proceedings of the USENIX Security symposium, Vancouver, BC, Canada, 16–18 August 2017.
15. Blazytko, T.; Bishop, M.; Aschermann, C.; Cappos, J.; Schlögel, M.; Korshun, N.; Abbasi, A.; Schweighauser, M.; Schinzel, S.; Schumilo, S. GRIMOIRE: Synthesizing structure while fuzzing. In Proceedings of the 28th USENIX Security Symposium (USENIX Security 19), Santa Clara, CA, USA, 14–16 August 2019; pp. 1985–2002.
16. Wang, J.; Chen, B.; Wei, L.; Liu, Y. Superion: Grammar-aware greybox fuzzing. In Proceedings of the 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), Montreal, QC, Canada, 25–31 May 2019; pp. 724–735.
17. Padhye, R.; Lemieux, C.; Sen, K.; Papadakis, M.; Le Traon, Y. Semantic fuzzing with zest. In Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, Beijing, China, 15–19 July 2019; pp. 329–340.
18. Gan, S.; Zhang, C.; Qin, X.; Tu, X.; Li, K.; Pei, Z.; Chen, Z. Collafl: Path sensitive fuzzing. In Proceedings of the 2018 IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 21–23 May 2018; pp. 679–696.
19. Odena, A.; Olsson, C.; Andersen, D.; Goodfellow, I. Tensorfuzz: Debugging neural networks with coverage-guided fuzzing. In Proceedings of the International Conference on Machine Learning, Long Beach, CA, USA, 10–15 July 2019; pp. 4901–4911.
20. Gao, Z.; Dong, W.; Chang, R.; Wang, Y.J.C.; Practice, C. Experience. Fw-fuzz: A code coverage-guided fuzzing framework for network protocols on firmware. *Concurr. Comput. Pract. Exp.* **2022**, *34*, e5756. [CrossRef]
21. Peng, H.; Shoshitaishvili, Y.; Payer, M. T-Fuzz: Fuzzing by program transformation. In Proceedings of the 2018 IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 20–24 May 2018; pp. 697–710. [CrossRef]
22. Stephens, N.; Grosen, J.; Salls, C.; Dutcher, A.; Wang, R.; Corbetta, J.; Shoshitaishvili, Y.; Kruegel, C.; Vigna, G. Driller: Augmenting fuzzing through selective symbolic execution. In Proceedings of the NDSS, San Diego, CA, USA, 21–24 February 2016; pp. 1–16.
23. Shoshitaishvili, Y.; Wang, R.; Hauser, C.; Kruegel, C.; Vigna, G. Firmalice-automatic detection of authentication bypass vulnerabilities in binary firmware. In Proceedings of the NDSS, San Diego, CA, USA, 7 February 2015; pp. 1.1–8.1.

24. Chipounov, V.; Kuznetsov, V.; Candea, G. S2E: A platform for in-vivo multi-path analysis of software systems. *Acm Sigplan Not.* **2011**, *46*, 265–278. [CrossRef]
25. Cha, S.K.; Avgerinos, T.; Rebert, A.; Brumley, D. Unleashing mayhem on binary code. In Proceedings of the 2012 IEEE Symposium on Security and Privacy, San Francisco, CA, USA, 20–23 May 2012; pp. 380–394.
26. Godefroid, P.; Levin, M.Y.; Molnar, D. SAGE: Whitebox fuzzing for security testing. *Commun. ACM* **2012**, *55*, 40–44. [CrossRef]
27. Yun, I.; Lee, S.; Xu, M.; Jang, Y.; Kim, T. QSYM: A practical concolic execution engine tailored for hybrid fuzzing. In Proceedings of the 27th USENIX Security Symposium (USENIX Security 18), Baltimore, MD, USA, 15–17 August 2018; pp. 745–761.
28. Wang, M.; Liang, J.; Chen, Y.; Jiang, Y.; Jiao, X.; Liu, H.; Zhao, X.; Sun, J. SAFL: Increasing and accelerating testing coverage with symbolic execution and guided fuzzing. In Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings, Gothenburg, Sweden, 27 May 2018—-3 June 2018; pp. 61–64.
29. Böhme, M.; Pham, V.-T.; Nguyen, M.-D.; Roychoudhury, A. Directed greybox fuzzing. In Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, Dallas, TX, USA, 30 October 30–3 November 2017; pp. 2329–2344.
30. Chen, H.; Xue, Y.; Li, Y.; Chen, B.; Xie, X.; Wu, X.; Liu, Y. Hawkeye: Towards a desired directed grey-box fuzzer. In Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, Toronto, ON, Canada, 15–19 October 2018; pp. 2095–2108.
31. Coppik, N.; Schwahn, O.; Suri, N. Memfuzz: Using memory accesses to guide fuzzing. In Proceedings of the 2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST), Xi'an, China, 22–27 April 2019; pp. 48–58. [CrossRef]
32. Nguyen, M.-D.; Bardin, S.; Bonichon, R.; Groz, R.; Lemerre, M. Binary-level Directed Fuzzing for Use-After-Free Vulnerabilities. In Proceedings of the RAID, San Sebastian, Spain, 14–15 October 2020; pp. 47–62.
33. Wang, H.; Xie, X.; Li, Y.; Wen, C.; Li, Y.; Liu, Y.; Qin, S.; Chen, H.; Sui, Y. Typestate-guided fuzzer for discovering use-after-free vulnerabilities. In Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, Seoul, Republic of Korea, 27 June 202–19 July 2020; pp. 999–1010.
34. Medicherla, R.K.; Komondoor, R.; Roychoudhury, A. Fitness guided vulnerability detection with greybox fuzzing. In Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops, Seoul, Republic of Korea, 27 June 202–19 July 2020; pp. 513–520.
35. Chen, J.; Diao, W.; Zhao, Q.; Zuo, C.; Lin, Z.; Wang, X.; Lau, W.C.; Sun, M.; Yang, R.; Zhang, K. IoTFuzzer: Discovering Memory Corruptions in IoT Through App-based Fuzzing. In Proceedings of the NDSS, San Diego, CA, USA, 18–21 February 2018.
36. You, W.; Zong, P.; Chen, K.; Wang, X.; Liao, X.; Bian, P.; Liang, B. Semfuzz: Semantics-based automatic generation of proof-of-concept exploits. In Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, Abu Dhabi, United Arab Emirates, 2–6 April 2017; pp. 2139–2154.
37. Wang, W.; Sun, H.; Zeng, Q. Seededfuzz: Selecting and generating seeds for directed fuzzing. In Proceedings of the 2016 10th International Symposium on Theoretical Aspects of Software Engineering (TASE), Shanghai, China, 17–19 July 2016; pp. 49–56.
38. Jain, V.; Rawat, S.; Giuffrida, C.; Bos, H. TIFF: Using input type inference to improve fuzzing. In Proceedings of the 34th Annual Computer Security Applications Conference, San Juan, PR, USA, 3–7 December 2018; pp. 505–517.
39. Lemieux, C.; Sen, K. Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage. In Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, Virtual Event, 18–21 February 2018; pp. 475–485.
40. You, W.; Wang, X.; Ma, S.; Huang, J.; Zhang, X.; Wang, X.; Liang, B. Profuzzer: On-the-fly input type probing for better zero-day vulnerability discovery. In Proceedings of the 2019 IEEE symposium on security and privacy (SP), San Francisco, CA, USA, 19–23 May 2019; pp. 769–786.
41. Gan, S.; Zhang, C.; Chen, P.; Zhao, B.; Qin, X.; Wu, D.; Chen, Z. GREYONE: Data Flow Sensitive Fuzzing. In Proceedings of the USENIX Security Symposium, Boston, MA, USA, 12–14 August 2020; pp. 2577–2594.
42. Lattner, C.; Adve, V. LLVM: A compilation framework for lifelong program analysis & transformation. In Proceedings of the International Symposium on Code Generation and Optimization, San Jose, CA, USA, 20–24 March 2004; pp. 75–86.
43. Serebryany, K.; Bruening, D.; Potapenko, A.; Vyukov, D. AddressSanitizer: A fast address sanity checker. In Proceedings of the Usenix Conference on Technical Conference, Boston, MA, USA, 13–15 June 2012.
44. Openjpeg. An Open-Source JPEG 2000 Codec Written in C Language. Available online: https://github.com/uclouvain/openjpeg (accessed on 19 February 2023).
45. Jasper. Image Processing/Coding Tool Kit. Available online: https://www.ece.uvic.ca/~frodo/jasper (accessed on 19 February 2023).
46. GNU Binutils. Acollection of Binary Tools. Available online: https://www.gnu.org/software/binutils/ (accessed on 19 February 2023).