



Article

BugMiner: Mining the Hard-to-Reach Software Vulnerabilities through the Target-Oriented Hybrid Fuzzer

Fayozbek Rustamov ^{1,2} , Juhwan Kim ^{1,2}, Jihyeon Yu ¹, Hyunwook Kim ¹ and Joobeom Yun ^{1,*} 

¹ Department of Computer and Information Security, Sejong University, 209 Neungdong-ro, Gwangjin-gu, Seoul 05006, Korea; ffrustamov@sju.ac.kr (F.R.); juhwan@sju.ac.kr (J.K.); yjhy8783@sju.ac.kr (J.Y.); hyunwook711@naver.com (H.K.)

² Department of Convergence Engineering for Intelligent Drone, Sejong University, 209 Neungdong-ro, Gwangjin-gu, Seoul 05006, Korea

* Correspondence: jbyun@sejong.ac.kr

Abstract: Greybox Fuzzing is the most reliable and essentially powerful technique for automated software testing. Notwithstanding, a majority of greybox fuzzers are not effective in directed fuzzing, for example, towards complicated patches, as well as towards suspicious and critical sites. To overcome these limitations of greybox fuzzers, Directed Greybox Fuzzing (DGF) approaches were recently proposed. Current DGFs are powerful and efficient approaches that can compete with Coverage-Based Fuzzers. Nevertheless, DGFs neglect to accomplish stability between usefulness and proficiency, and random mutations make it hard to handle complex paths. To alleviate this problem, we propose an innovative methodology, a target-oriented hybrid fuzzing tool that utilizes a fuzzer and dynamic symbolic execution (also referred to as a concolic execution) engine. Our proposed method aims to generate inputs that can quickly reach the target sites in each sequence and trigger potential hard-to-reach vulnerabilities in the program binary. Specifically, to dive deep into the target binary, we designed a proposed technique named BugMiner, and to demonstrate the capability of our implementation, we evaluated it comprehensively on bug hunting and crash reproduction. Evaluation results showed that our proposed implementation could not only trigger hard-to-reach bugs 3.1, 4.3, 2.9, 2.0, 1.8, and 1.9 times faster than Hawkeye, AFLGo, AFL, AFLFast, QSYM, and ParmeSan respectively but also scale to several real-world programs.

Keywords: directed fuzzing; hybrid fuzzing; concolic execution; software vulnerability; natural language processing



Citation: Rustamov, F.; Kim, J.; Yu, J.; Kim, H.; Yun, J. BugMiner: Mining the Hard-to-Reach Software Vulnerabilities through the Target-Oriented Hybrid Fuzzer. *Electronics* **2021**, *10*, 62. <https://doi.org/10.3390/electronics10010062>

Received: 1 December 2020

Accepted: 25 December 2020

Published: 31 December 2020

Publisher's Note: MDPI stays neutral with regard to jurisdictional claims in published maps and institutional affiliations.



Copyright: © 2020 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<https://creativecommons.org/licenses/by/4.0/>).

1. Introduction

The development of modern information technology is accompanied by adverse events such as industrial spyware, computer crimes, unauthorized access, and modification or loss of confidential information. The main reason for this is the presence of a vulnerability in the software. According to Edgescan's [1] vulnerability stats report 2019, it was discovered that network vulnerabilities increased slightly from 73% to 81%, and 27% of discovered application vulnerabilities decreased to 19%. Despite the fact that the Internet has a lot of vulnerability density, the application layer has the majority of high and critical risk exposures residing.

Software bugs have emerged as the underpinning reason for dangers to the safety of virtual life. Defined in RFC 2828 [2], a software flaw is an error or faintness in a system's layout, application, or process and control that could break the system's security policy. Attacking the system by using these bugs, specifically on 0-day vulnerabilities, can bring serious damages. Therefore, primarily hunting bugs is vital in the bug supervision procedure.

Fuzzing is the most reliable technique for automated software testing that feeds programs with random input and detects vulnerabilities critical to security [3]. The currently

increased application of fuzzing [4] in either academia and commerce, such as Microsoft's Springfield [5] and Google's OSS-FUZZ [6], demonstrates its aptitude in discovering numerous sorts of vulnerabilities in real-life programs. It is viewed as the most efficient and upgradable, which affords several seeds to the Program Under Test (PUT) and displays the nonstandard actions such as 0-day vulnerabilities, buffer or heap overflow, etc., [7]. Meanwhile, the suggested fuzzing has won preeminence in industry and academia. It also advanced into various forms of fuzzers for dissimilar testing situations. Fuzzing can be categorized as blackbox, whitebox, or greybox [8], rendering to their consciousness of the core structure of the PUT. Lately, greybox fuzzers have been widely utilized and recognized to be actual. Specifically, American Fuzzy Loop (AFL) [9] and its sources [10–13] receive significant percentages of attention.

While fuzzing can be extraordinarily powerful, it presents drawbacks at the same time. It has been unproductive at detecting hard-to-reach vulnerabilities deep inside programs [14], for the majority of randomly mutated new seeds fail to gratify the constraints that illustrate proper seeds and later fail to execute the rest of the code. Inspired by the necessity to find hard-to-reach bugs, scholars have implemented directed fuzzing [14–16].

Directed Grey-box Fuzzing (DGF) [11,17] is designed to fuzz on selected suspicious target locations, with programs to a variety of safety perspectives: (1) *bug reproduction* [17,18], for example, if a JavaScript engine for embedded systems called MJS [19] discovers a bug on MSP432 ARM platform that bug may arise within the conforming code for other platforms. In such a circumstance, the fuzzing must be directed to detect the vulnerability at these places, (2) *patch testing* [20,21], when a bug is fixed, the programmers are required to investigate whether the patch entirely fixes the bug. This involves fuzzing to emphasize its endeavors on those fixed codes. In these two situations, the fuzzing needs to be directed at attaining particular user-indicated target sites in the PUT. According to the application, the target sites are originated from static analysis reports, bug stack traces, or patches. Nevertheless, models of the DGF generating and parsing graphs and calculating distances in the instrumentation stage are very tedious. If those who use these DGF tools are sensitive to time consumption or have restricted computing resources, the above-mentioned might be extremely difficult. Consequently, such users will choose a lightweight and frivolous analysis approach, which can diminish the computing resource supplies and general analysis time.

We perform BugMiner as a new Target-Oriented Hybrid Fuzzer (TOHF) that attempts to overcome commonly described issues. BugMiner combines and directs two of the most efficient testing tools, including state-of-the-art fuzzing and Concolic Execution (CE) across input synchronization. The cardinal goal of BugMiner is to dive deeply into the program binary for mining the hard-to-reach vulnerabilities faster than other existing software testing tools. More specifically, the first step of the BugMiner is to identify unsafe functions and extract them from the Common Vulnerabilities and Exposures (CVE) [22]. To automatically extract unsafe functions, BugMiner utilizes Artificial Intelligence's text mining tool named Natural Language Processing (NLP) [23]. It is widely accepted that the advantages of natural language processing are innumerable. In addition, it is worth considering that we used NLP to mine bug report automatically and collect the vulnerable targets. Collected data were used in the next static analysis step to calculate the distance between the entry point of the program and the target function. Furthermore, we implemented a *BranchPruner* module to overcome the *path explosion* issue and enhance the hybrid fuzzing performance. After this process, the dynamic analysis starts to fuzz the program with fuzzing and CE. In addition, to dive deep faster into the program, we designed the dynamic analysis of BugMiner with an *input prioritization* module that categorizes the test-inputs.

To highlight the scalability and effectiveness of our proposed techniques, we comprehensively evaluated BugMiner on various benchmarks, such as *Lava-M dataset* [24], *Binutils* [25], *LibPNG* [26], *OpenSSL* [27] datasets, and eight popular real-world programs. In addition, we evaluated BugMiner's bug report analyzing method against 450 CVE bug reports. We compared the bug reproduction capability of our implementation against AFL [9], AFLFast [10], AFLGo [11], Hawkeye [17], QSYM [28], and ParmeSan [29]. Overall,

BugMiner outperformed all these software testing tools in bug reproduction and vulnerability exposure.

The vital contributions of this research include the following:

- This research presents a novel target-oriented hybrid fuzzing tool named BugMiner, which combines greybox fuzzing and concolic execution engine. It solves the constraints to execute the uncovered complex nested branches and generates more effective test-inputs to dive deep into the program.
- A set of novel methods is proposed to increase the efficiency of the bug hunting approach and decrease the time-consuming preprocessing. More specifically, we implemented a *Bug Report Analyzer* built on a Machine Learning tool named NLP. The bug report analyzer identifies vulnerable functions and extracts them from bug reports, then builds a specified target database that can be used in further steps. In addition, BugMiner provides three test-case pools (*T-Pool*) that create an opportunity to add newly generated inputs into different categories based on the priority.
- To enhance target-oriented hybrid fuzzing and overcome the *Path Explosion* problem of CE, we designed BugMiner with a *BranchPruner* module that gathers a set of branch addresses not related to the specified target. This allows BugMiner to trigger the hard-to-reach bug without excessive effort. We also significantly reduced the budget for bug hunting time than the existing approaches.
- We validated the effectiveness of BugMiner by carrying out several experiments with different datasets, including *LAVA-M*, *Binutils*, *LibPNG*, *OpenSSL*, and real-world programs. We compared the capability of BugMiner against the popular software testing tools, such as *AFL*, *AFLFast*, *AFLGo*, *Hawkeye*, *QSYM*, and *ParmeSan*. Comprehensive experiment results demonstrate that the proposed implementation can trigger hard-to-reach bugs faster than baseline tools and scale to several real-world programs. Furthermore, we provide dataset statistics for the comfort of readers.

The rest of this paper will proceed as follows. We introduce the background knowledge of this research in Section 2. Section 3 annotates our motivation for this research. In Section 4, we explain the proposed methodology to improve the efficiency of software testing. In Section 5, we depict the proposed method's implementation. Section 6 demonstrates various comparable experiments and examines the evaluation results. Finally, Section 7 concludes this research work.

2. Background

Fuzzing. Fuzzing is the most efficient technique to detect software vulnerabilities. There are blackbox, whitebox, and greybox fuzzing approaches in software testing. The simplest form of *Blackbox Fuzzing* produces random inputs to detect software vulnerabilities. The advantage of blackbox fuzzing is that it is straightforwardly well-matched with any program. On the opposite side of the range, there is *Whitebox Fuzzing* [30,31], utilizing heavyweight examination, for example, Symbolic Execution (SE), to produce test cases that trigger vulnerabilities, as opposed to blindly analyzing a massive variety of inputs. Practically, whitebox fuzzing is weak at compatibility problems or scalability in real-world applications. Recently, *Greybox Fuzzing*, which presents a medium position between whitebox and blackbox fuzzing, has become the most scalable and sensible fuzzing technique. Using the identical, the most adaptable methodology such as blackbox fuzzing is beneficial. Nevertheless, practical speaking strategies of greybox fuzzing procedures profit scalable and effective testing due to the lightweight mutation of seeds [32]. The famous state-of-the-art fuzzing tool is “American Fuzzy Lop” (AFL) [9] that utilizes performance outlining the information to mutate the seed. Meanwhile, such fuzzing techniques as VUzzer [33] and Angora [12], depending on active *Data-Flow Analysis* (DFA) to rapidly produce inputs that cause uncovered branches in the program binary, with the purpose of expanding code coverage.

Hybrid Fuzzing. Hybrid Fuzzing as the main topic has pulled in huge consideration and made noteworthy contributions to bug hunting. Hybrid fuzzing [28,34–36] naturally

combines fuzzing with SE or CE engine to deal with the insufficiency of both methodologies. The goal of the CE in a hybrid fuzzer is to enable the fuzzer to overcome narrow boundaries and better understand application logic. Driller [36] alternates the fuzzing with the CE during the program testing process. Moreover, the CE starts automatically when the fuzzing gets stuck. Another hybrid fuzzer, DigFuzz [37], evaluates the probability and selects the suitable paths by utilizing the “Monte Carlo” model. In order to recognize nested checksums, TaintScope [15] uses taint analysis, and then it starts to analyze the target with SE to generate the test cases. QSYM [28] improved the performance of hybrid fuzzing by merging CE and native execution. As a result, it showed high efficiency while testing real-world programs.

Coverage-based fuzzing or hybrid fuzzing are excellent standard approaches to acquire more paths and cover the more basic blocks, but discovering deep target bugs and analyzing patched bugs could be time-consuming with those undirected approaches. BugMiner tackles these issues by combining the fuzzer and *Target-Oriented Concolic Execution* (TOCE). To achieve high testing performance, TOCE takes inputs from the fuzzer, and, based on the paths, TOCE generates efficient inputs that can dive deep to trigger the hard-to-reach vulnerabilities.

Directed fuzzing. Directed fuzzing has been designed to guide fuzzing towards weak, vulnerable points in programs [38]. The instinct is that, compared with coverage-based fuzzing by directing fuzzing towards a particular target point within the program, the directed fuzzing can define precise vulnerabilities more quickly. As stated above, traditional directed fuzzing utilizes symbolic execution, which suffers from compatibility troubles or scalability.

Directed symbolic execution. SE plays a vital role in the directed fuzzer due to its path examination method. Therefore, the majority of proposed directed fuzzers [39–42] are whitebox fuzzing grounded on SE. Basically, they assemble symbolic path constraints while executing the program and run *constraint solving* modules to produce efficient inputs that can dive deeper into uncovered paths. As an example, Do et al. [39] apply an extended chaining method and information dependence analysis to construct event sequences main to the target locations and demonstrate directed concolic execution to produce aim-based test cases. To reach the specified target site and analyze programming patches, KATCH [20] associates a SE tool named KLEE [30] with numerous innovative heuristics. Another approach, BugRedux [18], uses a classification of application declarations as input and produces the seeds that trigger the bug. Directed symbolic execution alters the issue of reachability to constraint satisfiability issue and devotes the vast majority of execution time to heavyweight software examination and constraint solving, and, for this reason, this method is considered powerful [43]. Though the efficiency of the directed SE is high, to analyze the heavyweight real-world programs consumes significant time. On the other hand, BugMiner utilizes TOCE as a support to the fuzzing process that eases the incompetence of SE.

Directed Greybox Fuzzing. Within a time when the SE produces single test cases, a greybox fuzzer such as AFL [9] can create a few different test cases, so utilizing the greybox fuzzers can significantly improve the performance of the directed fuzzers. Hawkeye [17], Lolly [44], and AFLGo [11] are DGFs that emphasize the importance of arriving at the target sites in a program. AFLGo [11] is DGF that showed high efficiency in reaching the target by accepting a meta-heuristic to endorse the inputs. Specifically, it accepts target reachability challenges as a core issue and tries to diminish inputs’ distance to the target sites. Unfortunately, AFLGo converts the majority of program examination to the instrumentation stage in return for efficiency at runtime. Because the instrumentation phase of AFLGo calculates the distance between each node and the target position as well as measuring the seed distance to the target position, AFLGo needs to analyze the PUT’s call graph and the Control Flow Graph (CFG). It takes a long time to parse the graph and measure the distance at the instrumentation phase. Hawkeye [17] proposed an indirect function call that eases to calculate the distance and improved the performance of the

AFLGo to decrease the instrumentation time. One of the lightweight DGFs is Lolly [44], which analyses targets according to sequence coverage. However, the seed scheduling of Lolly is not as efficient as AFLGo. ParmeSan [29] proposed a novel sanitizer-guided fuzzer that increases the bug coverage automatically. To improve the distance evaluation accuracy, ParmeSan utilizes DFA in CFG construction time. It also uses DFA information in the bug detecting stage to achieve the low *Time-To-Exposure* (TTE). Zong et al. [45] proposed a new directed fuzzing tool named FuzzGuard that can trigger hard-to-reach vulnerabilities. More precisely, FuzzGuard is a Deep Learning based DGF that filter-out inefficient test-inputs that cannot arrive at a specified destination point; then, fuzzer executes the PUT with only productive test-inputs. As a result, the target software vulnerability can be triggered more quickly.

3. Motivation

A simplified example in Listing 1 shows problems regarding recent fuzzing tools that demonstrate our motivation. There is a *User-After-Free* (UAF) vulnerability because of a missing *exit ()* call, a typical root cause of these vulnerabilities, for example, CVE-2014-9296. The system reads a file, and the contents of this file are copied to a buffer. Precisely, after allocating the memory chunk pointed at through *m* (Line 17), *m_alias* and *m* become aliases (Line 20). The memory pointed to by each pointer is freed in the *vuln_func* (Line 15) function. The UAF vulnerability arises while the dereferenced memory is reclaimed through *m* (Line 24).

Listing 1. Motivation example.

```

1  #define BUF_SIZE 10
2  int *m, *m_alias;
3  char input[BUF_SIZE];
4  void vuln_func(int *m){
5      free(m);
6  } // here exit() function is missing
7  void func(){
8      if (input[1] == 'N'){
9          vuln_func(m);
10     } else {
11         // some common codes without vulnerabilities ...
12     }
13 }
14 int main(int argc, char *argv[]){
15     int file = open(argv[1], O_RDONLY);
16     read(file, input, BUF_SIZE);
17     m = malloc(sizeof(int));
18     if (input[0] == 'S'){
19         m_alias = malloc(sizeof(int));
20         m = m_alias;
21     }
22     func();
23     if (input[2] == 'U'){
24         *m = 1;
25     }
26     return 0;
27 }
```

Bug-triggering states. When the first three bytes of the seed are 'SNU', the UAF vulnerability is triggered. To determine this vulnerability immediately, fuzzer has to navigate the correct path through the *if* conditional statements in lines 8, 18, and 23 to deal with UAF's three events including *free*, *alloc* and *use*, respectively. It should be emphasized that this UAF vulnerability does not cause the program to crash, thus current greybox fuzzing tools without sanitization will not discover these kinds of vulnerabilities.

Coverage-based greybox fuzzing. Beginning with an empty input, AFL rapidly produces three new seeds such as 'SSS', 'NNN' and 'UUU' to activate separately the *free*,

alloc and *use* events. None of these inputs cause a memory error. Since the probability of producing the seed starting with ‘SNU’ from empty input is tremendously minor, the coverage-based fuzzing cannot be efficient here in monitoring a chain of UAF activities despite the fact that each distinct event is activated.

Directed Greybox Fuzzing. Provided a bug trace created by a lightweight instrumentation tool ASan [46], the DGF interrupts fuzzer from examining uncovered paths, such as the *else* part at *line 10* in function *func*, in case the circumstance at *line 8* is complicated. However, directed fuzzing tools have their misinterpretations. For instance, typical DGF seed selection methods support a seed that runs traces covering numerous sites on the target, rather than trying to arrive at these sites in the specified order. As an example, the standard DGF distances [11,17] for targets (S, N, U) do not distinguish between a seed S1 with the path S-N-U and other seed S2 with U-S-N. For one more example, Hawkeye proposed a power function that can allocate much energy to seeds where tracking does not arrive at the target location. This means that it might get lost in the toy example of the *else* part of *line 10*. AFLGo and Hawkeye cannot detect this kind of bug within 2 h, whereas our proposed BugMiner can detect this bug in less than 20 min.

Moreover, AFLGo is fruitful for patch test challenges and achieving high crash reproduction results, but it has a critical problem due to its distance calculating in the static analysis stage. AFLGo users suffer from excessive instrumentation time during preprocessing. As an example, in our comprehensive evaluation, AFLGo spent almost two hours compiling the *Libarchive* program and consumed more than three hours on compiling and instrumenting the *Popler* program. Performed observations encourage us to study a new, target-oriented hybrid fuzzing with high competence called BugMiner.

4. Proposed Methodology

This section will discuss the workflow design and methodology of BugMiner in detail. First, we express the overview of BugMiner in Section 4.1. Next, in Section 4.2, we explain on the *Bug Report Analyzer* approach that extracts unsafe functions from the bug reports. Section 4.3 details the *Static Phase* of BugMiner. Finally, Section 4.4 describes the *Dynamic Phase* of BugMiner, which contains the most up-to-date, fast, and powerful software testing approaches.

4.1. BugMiner Overview

This section provides a clear picture of the workflow of the proposed approach named BugMiner. Figure 1 depicts the overview of BugMiner, which includes three key components: bug report analyzer, static analysis, and dynamic analysis. The bug report analyzer demonstrates the feasibility of using NLP tools to automatically identify and extract the unsafe functions by analyzing descriptions of CVE. Extracted vulnerable functions are then placed in the *Targets* database for use in the static analysis process. The static analysis takes the program’s source code, and the specified target as input, then outputs the instrumented program binary, CFG, and *BranchPruner* data, which provides basic block level distance. In terms of dynamic analysis, the hybrid fuzzer which includes AFL, TOCE, and *Input Prioritization*, takes the target binary, an initial seed, target sites, and *BranchPruner* information as the inputs. The main result of the dynamic analysis is the test-inputs that cause the program to crash.

4.2. Bug Report Analyzer

The bug report analyzer identifies and extracts vulnerable functions from bug reports. More precisely, unique vulnerabilities of the software after discovery are often shared with big communities. Since the CVE bug report is well-formatted, it is feasible to utilize NLP tools to extract vulnerability-related information (e.g., unsafe function name) for other target programs with some extra effort. CVE [22] provides a reference method for publicly known security exposures and vulnerabilities, publishing information such as vulnerability type, unsafe functions, and affected versions.

As mentioned above, we implemented the bug report analyzer module based on the NLP techniques. In addition, the working process of this module is straightforward and includes *information retrieval* and *vulnerable function extraction*. Figure 2 illustrates the workflow of the bug report analyzer. In order to analyze CVE bug reports and extract unsafe function names automatically, first, this method retrieves the information by utilizing *Sentence Segmentation*, *Tokenizing*, *Syntactic Parsing*, *Part-of-speech (PoS)*, and *Chunking* techniques. The output of the first step is the *Parse Tree* that recognizes the PoS tag of each word. Next, we reanalyze the parse tree in the vulnerable function name extraction step. In this step, we convert the parse tree to string to extract the vulnerable function names. To do this, we implemented three methods that can easily extract unsafe function names. Although all three approaches' exact purpose is homogeneous, their performance indicators and time consumption for extraction are different.

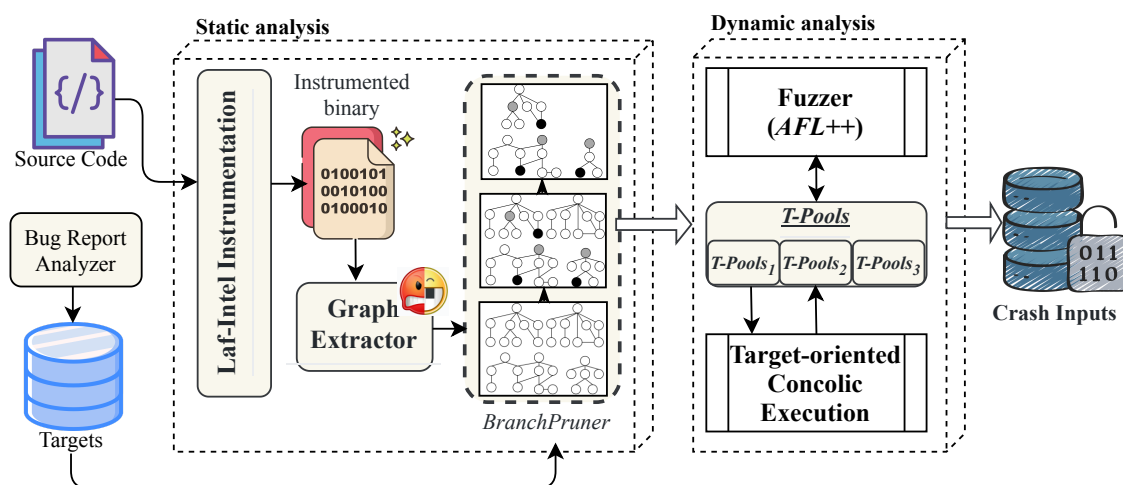


Figure 1. BugMiner overall workflow.

Neighbor approach. When one looks up CVE reports, it is easy to realize that the CVE has some fixed description patterns, such as:

- The AAA function in BBB file (other words) causes CCC (result) via a crafted EEE (method).
- FFF (reason) in the AAA function in BBB component (other words) could cause CCC (result).

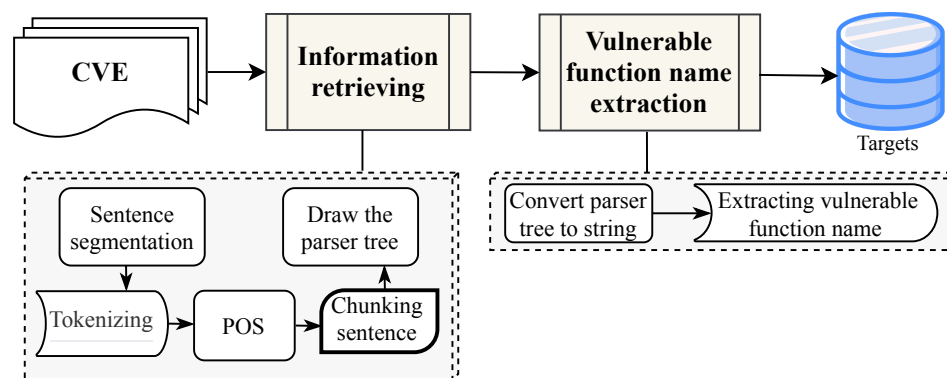


Figure 2. Overall workflow of the bug report analyzer.

One possible way is to summarize or learn the pattern of the CVE descriptions and then extract the possible function name for each category of the CVE description. To explain this approach clearly, we provide one bug report (CVE-2018-11237) as an example. Its vulnerability description is as follows: “An AVX-512-optimized implementation of the *mempcpy()*

function in the GNU C Library (aka glibc or libc6) 2.27 and earlier may write data beyond the target buffer, leading to a buffer overflow in *mempcpy_avx512_no_vzeroupper* [47]. In the given example, the vulnerable function *mempcpy* is located between the two words “the” and “function”. If the CVE descriptions are carefully looked up, it is clearly seen that 80% of CVE bug reports were written in this same structure. One of the easy ways to extract an unsafe function name in this method is that, first, we must find the word named “function”. Second, we have to check whether “the” article is located before the word “function” or not. If the first word is “the” and the third word is “function”, the unsafe function name is located between those two words.

Punctuation marks approach. This method is slightly different from other approaches and it is easier to use. In 15% of the CVE descriptions, punctuation marks are featured. Considering that the unsafe function and the punctuation mark come side by side, for example, *getcwd()* and *realpath()*, it is easy to extract the unsafe function name in this method. We first search for the punctuation mark “()”. Next, if this round bracket symbol is located inside the CVE description, the vulnerable function name is mainly located before the round bracket symbol.

Dictionary approach. This method differs significantly from other methods, despite its use of NLP techniques. To be more specific, words inside the CVE bug description have their own meanings in English but function names, such as *clntudp_call*, *posix_memalign*, *printf*, and *strcpy*, do not have any English meanings. In the third method, we extract unsafe function names with the help of an English dictionary which includes more than 20,000 words. This process is also straightforward. After retrieving the CVE bug description, the NLP tool parses it, then tokenizes and generates a parser tree. After that, the program automatically separates all *NP*, *NN* (noun) nodes of the parse tree, and then separated words are compared with the words in the English dictionary. If the words do not match the dictionary words, extracted function names are placed in the database.

In addition, Table 1 illustrates a part of the extracted unsafe functions which can cause an attack, and vulnerability, such as *format string*, *buffer overflow*, *multiple command injection*, and *DOS overflow*. For instance, *gets()* function is always vulnerable, so it seems reasonable for a static analysis tool to report all uses of *gets()*. The *strcpy()* function can be used safely but is often the source of buffer overflow vulnerabilities [48].

Table 1. Unsafe functions.

Type of Vulnerability	Unsafe Function Name
Format string	<i>dprintf</i> , <i>vasprintf</i> , <i>vsprintf</i> , <i>asprintf</i> , <i>vdprintf</i> , <i>snprintf</i> , <i>fprintf</i> , <i>printf</i> , <i>sprintf</i> , <i>vfprintf</i> , <i>vprintf</i> , <i>vsprintf</i> , <i>syslog</i> , <i>vsyslog</i>
Buffer overflow	<i>strcpy</i> , <i>stpcpy</i> , <i>strecpy</i> , <i>strcat</i> , <i>streadd</i> , <i>sprint</i> , <i>vprintf</i> , <i>gets</i> , <i>fscanf</i> , <i>vscanf</i> , <i>vfscanf</i> , <i>realpath</i> , <i>wcscpy</i> , <i>wcpcpy</i> , <i>mempcpy</i> , <i>getwd</i>
Multiple command injection	<i>popen</i> , <i>fexecve</i> , <i>execl</i> , <i>execvp</i> , <i>execvpe</i> , <i>system</i> , <i>execve</i> , <i>execv</i> , <i>execl</i> , <i>execlp</i>
DOS overflow	<i>memmove</i> , <i>glob</i> , <i>getaddrinfo</i> , <i>getbyname</i> , <i>gethostbyname</i> , <i>getifaddrs</i>

4.3. Static Analysis

The static analysis determines the software vulnerability by checking the program without execution. The static analysis examines the application in a variety of ways, although its analysis source code is straightforward and can be tested even if there is a defect in writing the program. In this way, utilizing static analysis gives us a chance to make assertions of pretty much all conceivable program executions as opposed to simply the experiment execution. From a security perspective, this is a critically preferred standpoint [48].

Graph Extractor. Directed fuzzers are dependent on statically generated CFGs for distance calculation. To achieve accurate distance, we first instrument the program and construct the CFG. The CFG is defined as a graph of basic code blocks, and it helps to find the path to the bug location. With the assistance of lightweight angr [49], we build the CFG. Angr generates the CFG relying statically on the analysis of the target program, starting from every function block, and searching the jump edges in the graph.

Branch Pruner. In target-oriented testing, it is pointless to analyze paths other than the path that leads to the specified target location, which can cause inefficient completion of the software testing process. This, in turn, complicates the bug hunting process and also causes path explosion issues. Therefore, to overcome these challenges, we implemented a straightforward method named *BranchPruner* that includes *ShortestPathFinder* and *BranchPruner* modules. To get the branch pruner locations, first, we need to identify the shortest path from the PUT's entry point to the specified target. The *ShortestPathFinder* gets each control flow and specified target site to calculate the inter-procedural distance for each node. The use of the target sites allows the *ShortestPathFinder* module to seek their strings in the recovered CFG. Specifically, it disassembles the PUT binary to identify the address of the specified target function. If it gets a specified target function address, it highlights this address as a target destination in the recovered CFG. For example, if a vulnerable function such as *gets()* is found, the branch address of this function can be a target destination point. After that, *ShortestPathFinder* gains addresses of basic blocks appeal to a target function and repetitively gather addresses of parent basic blocks until reaching the PUT entry point. When the *ShortestPathFinder* process is complete, the *BranchPruner* process begins. It collects the addresses of all branches not related to the shortest path. The collected branch pruning data are sent for dynamic analysis, and TOCE utilizes this data to inform the fuzzing with the right path to the target location.

The workflow of the branch pruner is illustrated in Algorithm 1. It takes the target program binary *PUT*, the CFG of the target program, and the specified target function S_{TF} from the target database as inputs. It returns branch pruning addresses B_{PA} , specified target branch address S_{TB} , and entry point of the PUT (E_p) as outputs. The branch pruning process starts from loading the *PUT* and identifying the program's entry point (lines 2–3). Then, it loads the CFG to extract all branch addresses (lines 4–5). Next, it disassembles the target program T_p to get the branch address of the S_{TF} . After the S_{TB} address is identified (line 7), it finds the shortest path and collects all branch addresses related to the path (line 8). Note that we utilized the *Dijkstra* [50] algorithm for finding the shortest path. The basic blocks that are not related to the *ShortestPathList* are put into the branch pruning addresses (B_{PA}). Otherwise, the process continues (lines 9–15).

4.4. Dynamic Analysis

Comparing to static analysis, dynamic analysis analyzes a program when it is running. Generally speaking, dynamic analysis allows the tester to see much more about the program, and it captures the concrete behavior of the software during a natural execution. There are different dynamic analysis approaches such as the fuzzer, SE, CE, and the hybrid fuzzer. Due to the effectiveness of hybrid fuzzing tools, they are currently widely used to test software applications.

Hybrid Fuzzer. Hybrid fuzzing as a research topic has gained enough popularity and made great contributions to software vulnerability detection. For instance, almost all the winners in the DARPA Cyber Grand Challenge [51] employed hybrid fuzzing tools. In comparison with plain fuzzing, hybrid fuzzing features and extra symbolic or concolic execution provide an opportunity to analyze the fuzzed paths, deal with the path conditions, and attempt to discover new uncovered paths. BugMiner combines AFL++ [52] and TOCE engine, which let us dive deeper into the program binary. These software testing approaches provide three seed queues with various priorities and add new efficient test inputs to the seed pool. In the following paragraphs, we explain each component of the BugMiner dynamic analysis.

Fuzzing approach. Although there is a number of effective fuzzing tools such as AFL [9], AFLFast [10], and LibFuzzer [53], we prefer to utilize AFL++ [52] fuzzer. The reason for our preference is that this approach is highly valued due to the efficiency and integration of the outstanding program bug hunting technique features [54]. With these features, AFL++ can be a more distinctive and proficient fuzzer than other modern fuzzers. The fuzzer receives the initial inputs to fuzz the PUT and stores a newly generated input to the $T\text{-Pool}_1$ queue. On the other hand, the TOCE analyzes the PUT with the inputs from $T\text{-Pool}_1$ queue, then generates new effective inputs by solving constraints and inserts input into $T\text{-Pool}_2$ queues. As a result, the fuzzer can cover deeply hidden paths by using inputs from the $T\text{-Pool}_2$ queue.

Algorithm 1 Branch pruning

INPUT: PUT (*Target Program Binary*) & CFG (*Control Flow Graph*)

INPUT: Specified Target Function (S_{TF})

```

1:  $B_{PA} \leftarrow \emptyset$ 
2:  $T_P \leftarrow \text{Load}(\text{PUT})$ 
3:  $E_P \leftarrow T_P.\text{findSymbol}('main').\text{addr}$ 
4: Graph  $\leftarrow \text{Load}(\text{CFG})$ 
5: AllBranchAddr  $\leftarrow \text{ExtractAllBranchAddr}(\text{Graph})$ 
6: Dis  $\leftarrow \text{get}.\text{Disassemble}(T_P)$ 
7:  $S_{TB} \leftarrow \text{Dis}.\text{getBranchAddress}(S_{TF})$ 
8: ShortestPathList  $\leftarrow \text{Graph}.\text{getShortestPaths}(E_P, S_{TB})$ 
9: foreach BB  $\in$  AllBranchAddr do {
10:   if BB  $\notin$  ShortestPathList do {
11:      $B_{PA} \leftarrow B_{PA} \cup \text{BB}$ 
12:   } else {
13:     Continue
14:   }
15: }
```

OUTPUT: Branch Pruning Addresses (B_{PA});

OUTPUT: Specified Target Branch (S_{TB});

OUTPUT: Entry Point of the PUT (E_P);

Algorithm 2 illustrates the greybox fuzzing process of BugMiner and the seed prioritization method. The fuzzer takes the *PUT* instrumented program and *I* initial input which can be built manually. It outputs crashing inputs that trigger the bug and newly generated test-inputs. The software testing process continues until the budget is exceeded or the abort signal is triggered. An initial input *I* is selected (line 4). Next, energy is assigned to the input (line 5), and then the mutation process of the fuzzer starts for the test-input (line 7). The fuzzer starts to execute the *PUT* with the mutated test-input T_c' . If *PUT* gets crashed by generated input T_c' , this test-input will be stored into the crash inputs C_I (line 8–9). Otherwise, if the generated test-input has a new branch coverage, it will be put into the $T\text{-Pool}_1$ (lines 10–11). If a test-input T_c' that cannot crash the *PUT* does not have any new branch coverage, it will be stored into the $T\text{-Pool}_3$ (line 13).

Input Prioritization. Not all the inputs are equally important for the software vulnerability testing tools. Unfortunately, the fuzzer also generates useless seeds that fail to cover new paths. Consequently, this makes the fuzzer time-consuming and reduces its efficiency. To mitigate this problem, BugMiner provides three test-case pools ($T\text{-Pool}$) that create an

opportunity to add newly generated inputs into different categories based on the priority. The inputs in the $T\text{-Pool}_1$ will be taken first, followed by inputs in the $T\text{-Pool}_2$, lastly $T\text{-Pool}_3$. The features of the inputs in the three priority $T\text{-Pools}$ are described as follows. Input in the $T\text{-Pool}_1$: the newly generated input should provide new coverage. Input in the $T\text{-Pool}_2$: TOCE should generate new input and provide a new coverage, unlike inputs that are stored in $T\text{-Pool}_1$. Input in the $T\text{-Pool}_3$: other seeds remain the least important and do not provide a new coverage.

Algorithm 2 Greybox fuzzing and seed prioritization

INPUT: PUT (Instrumented program)

INPUT: I (Initial Input)

```

1:  $C_I \leftarrow \emptyset$ 
2:  $T\text{-Pool}_1$  &  $T\text{-Pool}_3 \leftarrow \emptyset$ 
3: while TimeoutNotExhausted do {
4:    $T_c \leftarrow \text{SelectInput}(I)$ 
5:    $E \leftarrow \text{AssignEnergy}(T_c)$ 
6:   for  $i \leftarrow 0$  to  $E$  do {
7:      $T_c' \leftarrow \text{mutateInput}(T_c)$ 
8:     if  $T_c'$  crashes  $PUT$  {
9:        $C_I \leftarrow C_I \cup T_c'$ 
10:    } else if InputWithNewCoverage ( $T_c'$ ) == True {
11:       $T\text{-Pool}_1 \leftarrow T\text{-Pool}_1 \cup T_c'$ 
12:    } else {
13:       $T\text{-Pool}_3 \leftarrow T\text{-Pool}_3 \cup T_c'$ 
14:    }
15:  }
16: }
```

OUTPUT: Bug-triggering crash inputs C_I ;

OUTPUT: Newly generated test-inputs $T\text{-Pool}_1$ & $T\text{-Pool}_3$;

Target-oriented concolic execution. It is true that CE plays a key role in the software testing process. It picks partially suited seeds as input, expecting to produce effective seeds with big coverage. In contrast, the fuzzer produces inputs by randomly mutating them, and although effective inputs that can cover more paths are mutated by the fuzzer, it does not ensure that the newly produced inputs reach the target site. CE collects the input execution trace data and then creates inputs to execute uncovered paths through constraints solution. Figure 3 illustrates sample execution paths covered by the CE approach. All these covered basic blocks build a *concolic execution tree*. More specifically, the CE approach tries to cover all the program basic blocks and constructs the completed path tree. However, if the program is heavyweight or it has a significant number of branches to cover, tremendous concolic runs will be required or the software testing process will stop without success. This issue is one of the biggest limitations of CE is called the *Path Explosion* problem [55]. For instance, the *grep* program contains more than 15K LOC and 8.200 basic blocks. Trying to cover all these basic blocks leads to a path explosion problem.

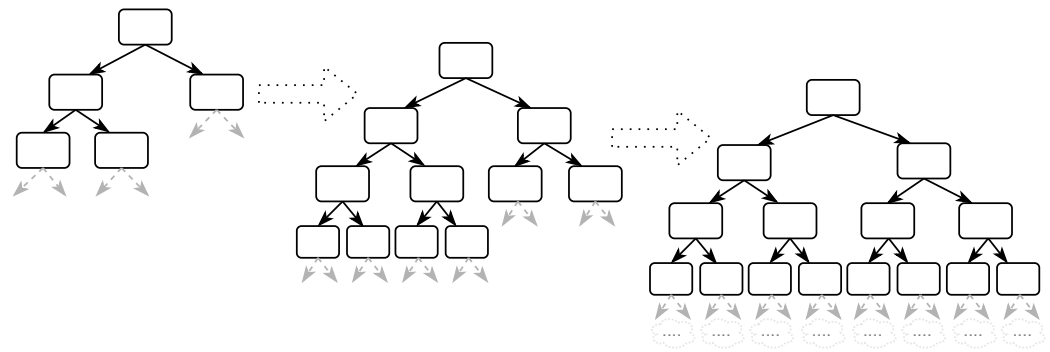


Figure 3. Construction of concolic execution tree.

To tackle this issue, we use the *BranchPruner* module information. Specifically, in target-oriented testing, executing the branches and paths that cannot lead to the specified target location is meaningless. The goal is to reach the deeply hidden specified target branch, solve the constraints based on the execution path, and generate efficient test input for the fuzzer. Therefore, pruning the branches that cannot lead to the specified target and then avoiding covering these unnecessary basic blocks makes the bug hunting process effective. We not only overcome the path explosion issue, but we can also decrease the software testing time by using *BranchPruner* information.

Algorithm 3 demonstrates BugMiner’s target-oriented concolic execution. TOCE takes target program PUT , test input T_I from $T-Pool_1$, branch pruning addresses B_{PA} , specified target branch address S_{TB} , and the entry point of the PUT E_P as input. It returns new target-oriented test-inputs as output. PUT is executed with T_I (line 2), and the entry point E_P is set (line 3). Then, it starts looping from the E_P until the specified destination-point S_{TB} (line 4). If B_{PA} is not contained *node* address (lines 5–6), then collected constraints are sent to the constraint solver engine to generate new test-inputs $NewTestInput$ until it reaches the specified target location S_{TB} (lines 7–9). Finally, each generated $NewTestInput$ is put into the $T-Pool_2$ queue.

Algorithm 3 Target-oriented concolic execution

INPUT: PUT (Target Program) & T_I (Test Input)

INPUT: B_{PA} (BranchPruningAddresses) & S_{TB} (SpecifiedTargetBranch)

INPUT: E_P (EntryPoint of the PUT)

```

1:  $T-Pool_2$  (NewTestInputs)  $\leftarrow \emptyset$ 
2:  $p \leftarrow PUT.execute(T_I)$ 
3:  $state \leftarrow p.entry(E_P)$ 
4: for  $state$  to  $S_{TB}$  do {
5:    $node \leftarrow getAddress(state)$ 
6:   if  $node \notin B_{PA}$  do {
7:     while  $node$  IsNotReachTo  $S_{TB}$  do {
8:        $Constraint \leftarrow p.take(node).getNeighbour()$ 
9:        $NewTestInput \leftarrow solve(Constraint)$ 
10:       $T-Pool_2 \leftarrow T-Pool_2 \cup NewTestInput$ 
11:    }
12:  }
13: }
```

OUTPUT: Generate new target-oriented test-inputs $T-Pool_2$ (NewTestInputs);

5. Implementation

BugMiner comprises three main components, *Bug Report Analyzer*, *Static Phase*, and *Dynamic Phase*. The following section aims to describe the implementation of these components.

Bug report analyzer. We implemented this model for the sake of extracting unsafe functions that are likely stored in bug reports. In this model, we employed the NLP machine learning tool, which enables the machine to realize the human language, operate, and analyze it. NLP allows us to get bug reports and extract the unsafe functions.

Static phase. The static phase involves the *Instrumentation*, the *Graph Constructor*, and *BranchPruner*. In the instrumentation, *LLVM Instrumentation* was employed with *afl-clang-fast++*. Specifically, the LLVM compiler transfers the target program's source code to LLVM IR (*Intermediate Representations*). The compiler environment variable is installed in *afl-clang-fast++* that encourages the building of the instrumented binary. The *Graph Constructor* module of BugMiner produces CFG. To do this, BugMiner utilizes a lightweight alias analysis, data flow tracking, and integrating pre-defined strategies provided by *angr*. In the *ShortestPathfinder* module, BugMiner acquires each control flow and target site to calculate the inter-procedural distance for every basic block. After the shortest path is found, we can gather addresses that will be pruned in further steps. We implemented the *BranchPruner* module in Python language based on the *Dijkstra* [50] algorithm, and, for parsing the CFG, we utilized the *networkx* library.

Dynamic Phase. The dynamic phase contains the fuzzer and the TOCE engine. More specifically, *AFL++ 2.60d* [54] was utilized as a fuzzer. In addition, we designed an *input priority* mechanism with three different *T-Pools*. The fuzzer selects an input according to its priority from a specific queue. In addition, to generate highly efficient target-oriented test inputs, we design and implement TOCE based on *angr*.

6. Performance Evaluation

This section is devoted to the evaluation of the prototype implementation in BugMiner and is also intended to examine the efficiency of our approach in the solution of bottlenecks.

6.1. Evaluation Setup

Research questions. In order to highlight the effectiveness of our approach, we address four practical research questions:

1. Is it beneficial to employ bug report analyzer and branch pruner components in our approach?
2. Do our suggested methods increase the software bug hunting process successfully?
3. How does our suggested dynamic strategy affect BugMiner's performance?
4. What is the role of BugMiner in achieving the deeply hidden target sites?

Evaluation dataset. To assess the effectiveness and usage of our TOHF, we carry out several experiments with different real-world programs.

- *Lava-M dataset* [24] is seen as an effective experimental vulnerable programs dataset to detect hard-to-reach bugs in the PUT. To evaluate the bug hunting tools, most of the software security researchers utilize this dataset.
- *Binutils* [25] is a combination of binaries used in the GNU/Linux system. This dataset also tested by a variety of famous research studies [10,11,17].
- *LibPNG* [26] is the official PNG reference library. It employs almost all PNG features, which have been widely used for over 23 years to evaluate software testing tools.
- *OpenSSL* [27] is a library for programs that provide communication security over computer networks as opposed to eavesdropping.
- Eight popular real-world applications can assist in evaluating the software bug detecting tools.

Experimental setup. All the experiments were evaluated on a machine with 16 GB of RAM and a 2.7 GHz Core i5-6400 processor. We used Ubuntu 16.04, 64 bit OS.

Evaluation tools. We made comparisons of BugMiner with the following fuzzing tools:

- *AFL* [9] is a commonly utilized state-of-the-art fuzzer.
- *AFLFast* [10] is an efficient greybox fuzzing tool that has optimized AFL with a new proposed power schedule algorithm.
- *AFLGo* [11] is modern, efficient DGF tool that relies on AFL. In comparison with AFL, it provides information about node distance.
- *Hawkeye* [17] is also a DGF tool that instruments the program to measure the distance of a certain test input to the target sites. Moreover, Hawkeye advances AFLGo by adding an indirect function call that facilitates the distance calculation of AFLGo.
- *QSYM* [28] is a current efficient undirected hybrid fuzzing tool that uses the CE to customize unnecessary computations in symbolic interpretations and improve the efficiency of constraint emulation.
- *ParmeSan* [29] refers to a “Sanitizer-Guided Greybox Fuzzing” (SGGF) tool which applies tripwire sanitization to direct the fuzzer and reveal violations earlier.

6.2. Bug Report Analyzing

We demonstrate the evaluation result of the bug report analyzer approaches that extracted vulnerable functions from CVE bug reports. To do this, we collected the CVEs demonstrated over the last seven years and obtained 450 CVEs that related to the GNU C library (glibc), Binutils [25], LibPNG [26], and OpenSSL [27]. There are also CVEs that contain vulnerabilities detected by well-known bug hunting tools [10,11,17,28] in the obtained 450 CVEs. These bug reports include the most common vulnerability types, such as *buffer overflow*, *buffer over-read*, *use-after-free*, *DOS*, *exec code overflow*, and others. Bug report analyzer, a part of the BugMiner, successfully extracted all 450 CVE unsafe functions.

Figure 4 illustrates the comparison of the bug report analyzing approaches. Figure 4a shows the accuracy of bug report analyzers. It can be clearly seen from the diagram that the Dictionary method reached the highest accuracy with 94%. However, this method is more likely to produce more *false-positive* results than the other methods. In addition, the average elapsed time of *Neighbor*, *Punctuation mark*, and *Dictionary* methods for bug report analysis was 1.7, 1.9, and 1.69 s, respectively. Figure 4b illustrates the number of extracted unsafe functions from 450 CVE bug reports. The Neighbor approach extracted 409 unsafe functions from 450 CVEs. Considering that the descriptions of other 41 CVEs bug structures are different, this approach can exactly identify and extract unsafe function names via pattern recognition if the description of the bug report structure is the same as “*the*” article + “*vulnerable function name*” + word of “*function*”. In the Punctuation marks approach, we extracted 41 CVEs’ vulnerable function names from 450 CVEs. However, as mentioned in the neighbor approach, the punctuation mark approach extracts unsafe functions based on CVE’s bug description structure. Due to the other 409 CVE reports, which do not contain punctuation marks, this method stayed at the lowest place.

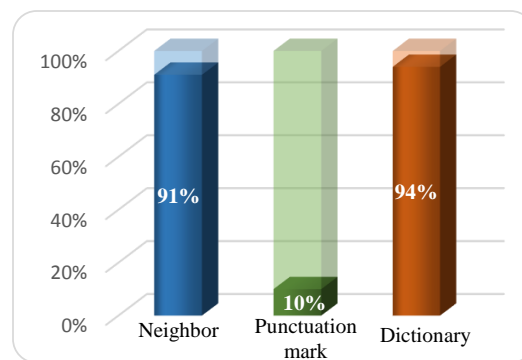
6.3. Bug Reproduction

Software users often suffer from vulnerable programs, which cause huge damages. Most software applications are implemented with bug reporting techniques that report to developers when the crash happens. However, bug reports are limited to providing input data that causes crashes, and it mainly includes call stacks. According to the reported call stacks, developers are required to fix bugs in the program.

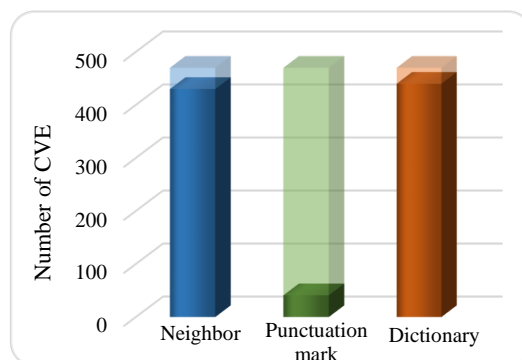
We evaluate the ability and efficiency of BugMiner’s guidance by comparing it with AFL, AFLFast, and QSYM. In this evaluation, we calculate and compare the average time of BugMiner and baseline approaches to reproduce triggered bugs in the LAVA-M dataset. The LAVA-M dataset consists of four buggy programs *md5sum*, *uniq*, *who*, and *base64*.

We selected this dataset because it contained many complicated bugs, and also most of the efficient fuzzing tools to evaluate the performance of their proposed approaches by testing this dataset. Additionally, Table 2 demonstrates the LAVA-M dataset details. We select two bugs for each PUT to reproduce these bugs. We then executed BugMiner and

the above-mentioned four fuzzers for three hours with the same seeds. Reported numbers illustrated in Table 3 are the average result of eight measurements. The first column indicates the dataset programs, and the second column shows TTE results, which calculates the duration of the fuzzing time until the specified vulnerability occurs. As can be seen from Table 3, BugMiner detected all target bugs faster than other competing fuzzers, except *bug 1* and *bug 2* of *who*. In a word, BugMiner is more efficient due to the fast test input generation compared to other state-of-the-art fuzzers, and it serves to increase time efficiency.



(a) Accuracy of bug report analyzers



(b) Extracted unsafe functions from 450 CVE

Figure 4. Evaluation of bug report analyzer approaches.

Table 2. LAVA-M dataset details.

Program	T/Bugs	Edges	Nodes	Instructions	Test Options
md5sum	57	1560	1013	7397	-c @@
uniq	28	1407	890	5285	@@
who	2136	3332	1831	84,648	@@
base64	44	1308	822	-	-d @@

Table 3. Bug reproduction on LAVA-M dataset.

Program	TTE (min)							
	AFL		BugMiner		AFLFast		QSYM	
	Bug 1	Bug 2	Bug 1	Bug 2	Bug 1	Bug 2	Bug 1	Bug 2
md5sum	14.25	13.51	10.49	09.53	15.07	13.59	14.08	12.56
uniq	29.32	31.09	18.35	20.41	29.09	30.56	28.45	30.01
who	67.32	52.32	51.30	47.53	59.28	51.39	50.12	46.23
base64	33.45	34.02	20.45	18.03	30.36	32.14	28.23	21.54

6.4. BugMiner vs. Directed Fuzzers

In this evaluation, we demonstrate the comparison results between our implementation and the state-of-the-art guided greybox fuzzing tools [11,17,29]. We reproduce a number of benchmarks covered by AFLGo and Hawkeye to represent the fares of BugMiner in a traditional directed setting. The CVE vulnerabilities that are utilized in this evaluation are given in Table 4. Note that the source code of Hawkeye is publicly unavailable, and, hence, we compare other fuzzers against results provided by Hawkeye authors. Furthermore, we also calculated the time execution spent on preprocessing analysis in each DGF to prove the efficiency of BugMiner’s static analysis.

Table 4. CVE vulnerabilities utilized in evaluations.

Program	CVE-ID	Type of Vulnerability
OpenSSL	CVE-2014-0160	Buffer Overflow
LibPNG	CVE-2011-2501	Buffer Overflow
LibPNG	CVE-2011-3328	Division by Zero
LibPNG	CVE-2015-8540	Buffer Overflow
Binutils	CVE-2016-4487	Invalid Write
Binutils	CVE-2016-4488	Invalid Write
Binutils	CVE-2016-4489	Invalid Write
Binutils	CVE-2016-4491	Stack Corruption
Binutils	CVE-2016-4492	Write Access Violation
Binutils	CVE-2016-4493	Write Access Violation

Table 5 shows a comparison of AFLGo, Hawkeye, ParmeSan, and BugMiner on bug reproduction of known vulnerabilities in *OpenSSL*, *LibPNG*, and *Binutils*. The first column indicates CVE identification number, the second column shows the fuzzing tool, and the next one is the number of fuzzing executions that successfully activated bugs. The TTE values are stored in the fourth column, and the last column presents the time consumed in preprocessing analysis. We repeatedly experimented with each CVE 20 times and illustrated the average results. In addition, each experiment ran for eight hours.

As indicated in Table 5, BugMiner shows better performance in all bug reproduction cases than the state-of-the-art directed fuzzers. It also performed 3.1, 4.3, and 1.9 times faster in the bug reproduction than Hawkeye, AFLGo, and ParmeSan, respectively. As shown in the last column, BugMiner spent less time on preprocessing analysis compared to other fuzzers. To sum up, BugMiner also highly decreases the preprocessing time by proposed *BranchPruner*, *ShortestPathFinder* methods, and improves the bug hunting performance.

6.5. Vulnerabilities Exposure

The directed fuzzing is the most prevalent tool that attempts to detect the bug in suspicious locations that could be vulnerable. To evaluate the capability of BugMiner’s exposure to vulnerability in real-world programs, we opted for the most common software listed in Table 6 [35]. We allocate 8 h to run each experiment, and we continuously tested 10 times for each.

Figure 5 illustrates TTE results in eight real-world programs. In this evaluation, we compare BugMiner’s bug detection ability with AFL, AFLGo, AFLFast, QSYM, and ParmeSan. It is obvious from the figure that BugMiner is more effective than other fuzzers as it detected a vulnerability in the *cert-basic* program that other software testing tools failed to find. In addition, our implementation spent less time to reach the specified target than other fuzzers.

In addition, to highlight the accuracy of BugMiner, we calculated the false positive rate (FPR) and the false negative rate (FNR) based on the FPR and FNR calculation method of FuzzGuard [45]. Figure 6 illustrates accuracy results of BugMiner on eight real-world programs. The average FPR and FNR of BugMiner for all specified targets are 0.89% and 0.021%, respectively. The reachability accuracy of test-inputs that produced by BugMiner

is 99.18% on average. According to the FuzzGuard [45] paper, it achieved 98.7% average accuracy by filtering out unreachable test-inputs. If we compare the accuracy of both techniques, we can see that our method has 0.48% slightly higher accuracy than FuzzGuard.

Table 5. Performance of BugMiner over directed fuzzers.

CVE ID	Fuzzer	Runs	TTE (min)	Pre-Process (min)
OpenSSL				
2014–0160	AFLGo	20	20.54	31.24
	Hawkeye	-	-	-
	ParmeSan	25	5.35	16.21
	BugMiner	20	3.25	6.09
LibPNG				
2011–2501	AFLGo	20	6.32	26.21
	Hawkeye	-	-	-
	ParmeSan	20	6.02	18.54
	BugMiner	20	4.29	9.01
2011–3328	AFLGo	20	42.01	29.39
	Hawkeye	-	-	-
	ParmeSan	20	31.56	22.45
	BugMiner	20	28.59	12.47
2011–8540	AFLGo	20	2.01	15.08
	Hawkeye	-	-	-
	ParmeSan	20	4.51	11.57
	BugMiner	20	3.12	4.23
CVE ID	Fuzzer	Runs	TTE (min)	Pre-Process (min)
Binutils				
2016–4487	AFLGo	20	6.09	27.45
	Hawkeye	20	2.57	-
	ParmeSan	20	1.05	21.35
	BugMiner	20	1.06	14.02
2016–4489	AFLGo	20	3.02	19.27
	Hawkeye	20	3.26	-
	ParmeSan	20	1.08	12.56
	BugMiner	20	1.09	3.59
2016–4491	AFLGo	5	385	59.54
	Hawkeye	9	312	-
	ParmeSan	10	71	25.21
	BugMiner	13	66	20.49
2016–4492	AFLGo	20	8.45	21.09
	Hawkeye	20	7.57	-
2016–4493	ParmeSan	20	2.56	10.24
	BugMiner	20	2.21	5.54

Table 6. Details of eight real-world programs.

Package	Target Programs	Version	Bug Type	Class Type	Basic Blocks	Edges	Input Format	Test Options	Source
binutils	readelf	2.30	BO	dev	21,249	31,086	binary	-a @@	[25]
binutils	objdump	2.30	HBO	dev	43,935	74,313	binary	-d @@	[25]
libksba	cert-basic	1.3.5	NPD	crypto	9958	14,120	crt	@@	[56]
libjpeg	djpeg	9c	NPD	image	4844	6776	jpg		[57]
libarchive	bsdtar	3.3.2	ML	archive	31,379	43,390	tar	@@	[58]
tcpdump	tcpdump	4.9.2	IO	net	33,743	48,791	pcap	-r/-nr @@	[59]
poppler	pdftohtml	0.22.5	UAF	doc	54,596	71,945	html/pdf	@@	[60]
poppler	pdftotext	0.22.5	BO	doc	49,867	62,391	pdf/txt		[60]

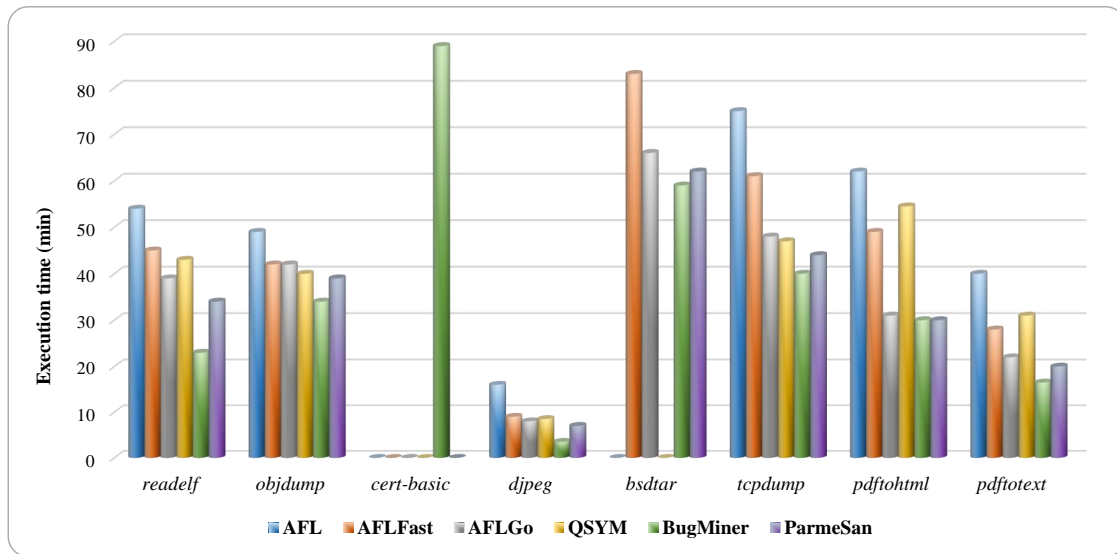


Figure 5. Evaluation results on eight real-world programs.

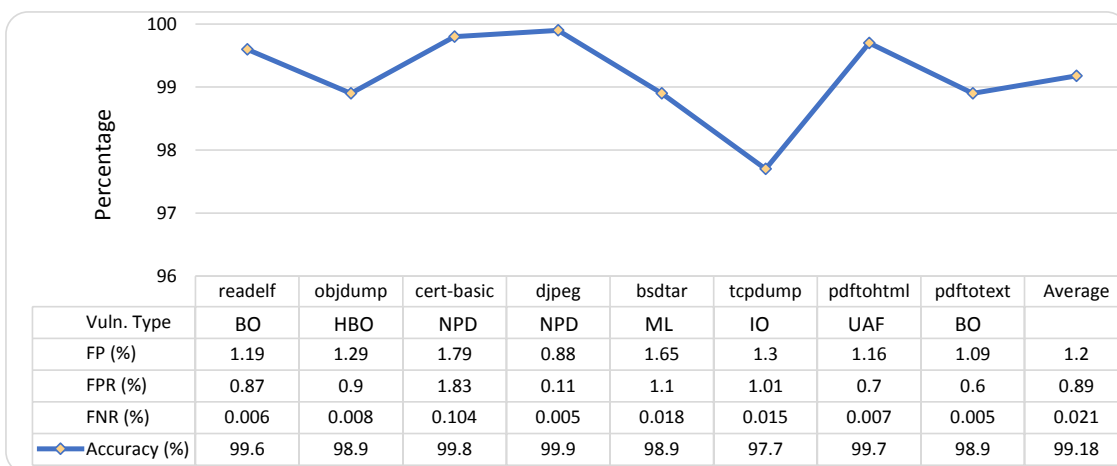


Figure 6. Accuracy results on eight real-world programs.

The reasons for the high accuracy of BugMiner and the low FPR, FNR are as follows. In general, if the number of constraints or complex nested checks in the path from the entry point to the specified target site is significantly higher, the reachability accuracy of the generated test-inputs will be less. In BugMiner, solving constraints and complex nested checks using TOCE ensured high accuracy. More precisely, the proposed TOCE generates productive test-inputs through a constraint solving engine and provides the fuzzer with these test-inputs. In addition, using *BranchPruner* data from static analysis, TOCE restricts exploring branch addresses that do not lead to the specified target site and focuses only on the path where the specified target is located. As a result, TOCE generates reachable test-inputs.

6.6. Replies to Research Questions

Based on the experiments illustrated in Tables 3 and 5, and Figure 5, they enable us to respond to the research questions given above.

1. We believe that *Bug Report Analyzer* and *BranchPruner* methods are worth applying. As we can see in Table 5, the *BranchPruner* method is a better option considering

the time cost while fuzzing. To be more precise, AFLGo, ParmeSan, and BugMiner spent an average of 28.6 min, 17.2 min, and 9.4 min, respectively, for preprocessing analysis. The proposed *ShortestPathFinder* reduces preprocessing time. Instead of making a specified target database manually, retrieving vulnerable functions from the bug reports automatically by utilizing the machine learning technology not only improves the performance of directed testing but also reduces preprocessing time.

2. BugMiner demonstrated better performance in finding bugs. It is clear from results in Tables 3 and 5, and Figure 5 that BugMiner can detect bugs 3.1, 4.3, 2.9, 2.0, 1.8, and 1.9 times faster than Hawkeye, AFLGo, AFL, AFLFast, QSYM, and ParmeSan, respectively.
3. The dynamic analysis methods utilized in BugMiner are highly effective. It is proven that BugMiner outperforms other fuzzing tools in all experiments we conducted. In particular, the experiment involving comparisons with the DGFs. Table 5 and Figure 5 indicate that the combination of fuzzing, TOCE, and input prioritization methods increase BugMiner's speed, which gives an advantage over DGFs, hybrid fuzzer, and greybox fuzzers. In addition, BugMiner achieved 99.18% accuracy on average.
4. Based on the results in Figure 5, we believe that BugMiner affords an opportunity to achieve specified targets rapidly. In addition, our implementation is scalable and can analyze both coreutil programs and real-world programs.

7. Conclusions

In this research, we propose a novel target-oriented hybrid fuzzing tool named BugMiner that combines fuzzing and TOCE that enhances the directed fuzzing process. We also propose a set of novel approaches to overcome the path explosion problem of CE and improve the effectiveness of the bug hunting process. More precisely, we implemented the machine-learning-based module *Bug Report Analyzer* to build a specified target database automatically, and it successfully extracted all 450 CVE unsafe functions. In addition, we reduced the preprocessing time using the *BranchPruner* method to 19.2 and 7.8 min compared to AFLGo and ParmeSan. In addition, we proposed the *Input Prioritization* module that categorizes the test-inputs to trigger the deeply hidden vulnerabilities.

We validated the scalability of BugMiner by carrying out several experiments with different datasets and real-world programs. It is obvious from experimental results that BugMiner is considered more effective than state-of-the-art DGFs, such as Hawkeye, AFLGo, and ParmeSan, in bug reproduction and preprocessing analysis. Particularly, BugMiner detects software vulnerabilities with considerably lower TTE than state-of-the-art fuzzers. In the bug reproduction, BugMiner performed 3.1, 4.3, 2.9, 2.0, 1.8, and 1.9 times faster than Hawkeye, AFLGo, AFL, AFLFast, QSYM, and ParmeSan, respectively. We believe that the approaches proposed by BugMiner, such as *Bug Report Analyzer*, *BranchPruner*, *Input Prioritization*, and TOCE can improve the performance of the software testing process. In the future, we aim to optimize BugMiner with deep learning methods to generate more efficient seeds that can increase the code coverage.

Author Contributions: F.R. contributed the ideas and wrote the paper; F.R. and J.K. designed and conducted the experiments; J.Y. (Jihyeon Yu) provided resources and database of vulnerable programs; H.K. performed bug report analysis and collected the target unsafe functions; J.Y. (Joobeom Yun) supervised the whole paper including paper organization and proofreading. All authors have read and agreed to the published version of the manuscript.

Funding: This research was supported by the MSIT (Ministry of Science and ICT), Korea, under the ITRC (Information Technology Research Center) support program (IITP-2020-2020-0-01602) supervised by the IITP (Institute for Information & Communications Technology Planning & Evaluation). In addition, this research was supported by the Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education (number NRF-2018R1D1A1B07047323).

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Not applicable.

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

Abbreviations

The following abbreviations are used in this manuscript:

CE	Concolic Execution
SE	Symbolic Execution
PUT	Program Under Test
DGF	Directed Grey-box Fuzzing
CVE	Common Vulnerabilities and Exposures
NLP	Natural Language Processing
TOHF	Target-Oriented Hybrid Fuzzer
TOCE	Target-Oriented Concolic Execution
DFA	Data-Flow Analysis
UAF	User-After-Free
CFG	Control Flow Graph
TTE	Time-To-Exposure
SGGF	Sanitizer-Guided Greybox Fuzzing
BO	Buffer Overflow
HBO	Heap Buffer Overflow
NPD	NULL Pointer Dereference
ML	Memory Leak
IO	Integer Overflow
FPR	False Positive Rate
FNR	False Negative Rate

References

1. Eoin, K. 2019 Vulnerability Statistics Report. Available online: <https://www.edgescan.com/wp-content/uploads/2019/02/edgescan-Vulnerability-Stats-Report-2019.pdf> (accessed on 29 April 2020)
2. Shirey, R. Internet Security Glossary. Available online: <https://tools.ietf.org/html/rfc2828> (accessed on 29 April 2020)
3. Takanen, A.; DeMott, J.; Miller, C.; Kettunen, A. *Fuzzing for Security Testing and Quality Assurance*, 2nd ed.; Artech House: London, UK, 2018; pp. 1–2. ISBN 9781608078509.
4. Valentin, J.; HyungSeok, H.; Choongwoo, H.; Sang, K.C.; Manuel, E.; Edward, J.S.; Maverick, W. The art, science, and engineering of fuzzing: A survey. *IEEE Trans. Softw. Eng.* **2019**. [CrossRef]
5. Microsoft. Project Springfield. Available online: <https://www.microsoft.com/en-us/security-risk-detection/> (accessed on 30 April 2020).
6. OSS-Fuzz: Continuous Fuzzing Framework for Open-Source Projects. Available online: <https://github.com/google/oss-fuzz/> (accessed on 30 April 2020).
7. Crawford, J.B. A Survey of Some Free Fuzzing Tools. Available online: <https://lwn.net/Articles/744269/> (accessed on 1 May 2020).
8. Chen, C.; Baojiang, C.; Jinxin, M.; Runpu, W.; Jianchao, G.; Wenqian, L. A systematic review of fuzzing techniques. *Comput. Secur.* **2018**, *75*, 118–137. [CrossRef]
9. Zalewski, M. American Fuzzy Lop (AFL), README. Available online: <http://icamtuf.coredump.cx/afl/> (accessed on 2 May 2020).
10. Böhme, M.; Pham, V.T.; Roychoudhury, A. Coverage-based Greybox Fuzzing as Markov Chain. In Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS), Vienna, Austria, 24–28 October 2016.
11. Böhme, M.; Pham, V.T.; Manh-Dung, N.; Roychoudhury, A. Directed Greybox Fuzzing (CCS '17). In Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, New York, NY, USA, 30 October–3 November 2017; pp. 2329–2344.
12. Chen, P.; Chen, H. Angora: Efficient fuzzing by principled search. In Proceedings of the IEEE Symposium on Security and Privacy 39th IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 20–24 May 2018; pp. 711–725.
13. Shuitao, G.; Chao, Z.; Xiaojun, Q.; Xuwen, T.; Kang, L.; Zhongyu, P.; Zuoning, C. CollAFL: Path Sensitive Fuzzing. In Proceedings of the 39th IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 20–24 May 2018; pp. 1–12.

14. Ganesh, V.; Leek, T.; Rinard, M.C. Taint-based directed whitebox fuzzing. In Proceedings of the 31st International Conference on Software Engineering, ICSE, Washington, DC, USA, 16–24 May 2009; pp. 474–484.
15. Wang, T.; Wei, T.; Gu, G.; Zou, W. TaintScope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection. In Proceedings of the IEEE Symposium on Security and Privacy, Berkeley/Oakland, CA, USA, 16–19 May 2010.
16. Vuagnoux, M. Autodafe: An act of software torture. In Proceedings of the 22nd Chaos Communications Congress, Berlin, Germany, 27–30 December 2005; pp. 47–58.
17. Hongxu, C.; Yinxiang, X.; Yuekang, L.; Bihuan, C.; Xiaofei, X.; Xiuheng, W.; Yang, L. 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.
18. Wei, J.; Alessandro, O. Bugredux: Reproducing field failures for in-house debugging. In Proceedings of the 34th International Conference on Software Engineering (ICSE), Zurich, Switzerland, 2–9 June 2012.
19. Andrew, F.; Tim, L.; Brendan, D.G.; Josh, B. The rodeo day to less-buggy programs. *IEEE Secur. Privacy* **2019**, *17*, 84–88.
20. Paul, D.M.; Cristian, C. Katch: High-coverage testing of software patches. In Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, Saint Petersburg, Russia, 18–26 August 2013.
21. Jiaqi, P.; Feng, L.; Bingchang, L.; Lili, X.; Binghong, L.; Kai, C.; Wei, H. 1dvul: Discovering 1-day vulnerabilities through binary patches. In Proceedings of the 2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), Portland, OR, USA, 24–27 June 2019.
22. What Is CVE, Its Definition and Purpose? Available online: <https://www.csoonline.com/article/3204884/what-is-cve-its-definition-and-purpose.html> (accessed on 21 May 2020).
23. Hardeniya, N.; Perkins, J.; Chopra, D. *Natural Language Processing: Python and NLTK*; Packt Publishing Ltd.: Birmingham, UK, 2016; pp. 96–178.
24. Dolan-Gavitt, B.; Hulin, P.; Kirda, E.; Leek, T.; Mambretti, A.; Robertson, W.; Ulrich, F.; Whelan, R. Lava: Large-scale automated vulnerability addition. In Proceedings of the 2016 IEEE Symposium on Security and Privacy (SP), IEEE, San Jose, CA, USA, 22–26 May 2016; pp. 110–121.
25. Binutils Source Code. Available online: <https://ftp.gnu.org/gnu/binutils/> (accessed on 23 March 2020).
26. LibPNG—A Library for Processing PNG Files. Available online: <http://www.libpng.org/pub/png/libpng.html> (accessed on 29 April 2020).
27. OpenSSL-Cryptography and SSL/TLS Toolkit. Available online: <https://ftp.openssl.org/source/old/1.0.1/> (accessed on 20 April 2020).
28. 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 Conference on Security Symposium, USENIX Association, Baltimore, MD, USA, 15–17 August 2018; pp. 745–761.
29. Osterlund, S.; Razavi, K.; Bos, H.; Giuffrida, C. ParmeSan: Sanitizer-guided Greybox Fuzzing. In Proceedings of the 29th USENIX Security Symposium, Boston, MA, USA, 12–14 August 2020.
30. Cadar, C.; Dunbar, D.; Engler, D. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation, USENIX Association, Berkeley, CA, USA, 8–10 December 2008; pp. 209–224.
31. Godefroid, P.; Levin, M.Y.; Molnar, D.A. Automated whitebox fuzz testing. In Proceedings of the Network and Distributed System Security Symposium (NDSS), Reston, VA, USA, 10–13 February 2008.
32. Phil, M. Search-based software test data generation: A survey: Research articles. *Softw. Test. Verif. Reliab.* **2004**, *14*, 105–156.
33. Rawat, S.; Jain, V.; Kumar, A.; Cojocar, L.; Giuffrida, C.; Bos, H. VUzzer: Application-aware evolutionary fuzzing. In Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS), San Diego, CA, USA, 26 February–1 March 2017.
34. Chen, Y.; Li, P.; Xu, J.; Guo, S.; Zhou, R.; Zhang, Y.; Wei, T.; Lu, L. Savior: Towards bug-driven hybrid testing. In Proceedings of the 2020 IEEE Symposium on Security and Privacy (SP), Los Alamitos, CA, USA, 18–21 May 2020.
35. Rustamov, F.; Kim, J.; Yun, J. DeepDiver: Diving into abysmal depth of the binary for hunting deeply hidden software vulnerabilities. *Future Internet* **2020**, *12*, 74. [[CrossRef](#)]
36. 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.
37. Zhao, L.; Duan, Y.; Yin, H.; Xuan, J. Send hardest problems my way: Probabilistic path prioritization for hybrid fuzzing. In Proceedings of the NDSS, San Diego, CA, USA, 24–27 February 2019.
38. Maria, C.; Peter, M.; Valentin, W. Guiding dynamic symbolic execution toward unverified program executions. In Proceedings of the 38th International Conference on Software Engineering (ICSE), Austin, TX, USA, 14–22 May 2016; pp. 144–155.
39. Do, T.; Fong, A.; Pears, R. Dynamic Symbolic Execution Guided by Data Dependency Analysis for High Structural Coverage. In Proceedings of the Evaluation of Novel Approaches to Software Engineering—7th International Conference (ENASE), Warsaw, Poland, 29–30 June 2012; pp. 3–15.
40. Ge, X.; Taneja, K.; Xie, T.; Tillmann, N. DyTa: Dynamic symbolic execution guided with static verification results. In Proceedings of the 33rd International Conference on Software Engineering (ICSE), Waikiki, HI, USA, 21–28 May 2011; pp. 992–994.
41. McMinn, P.; Holcombe, M. Evolutionary Testing Using an Extended Chaining Approach. *Evol. Comput.* **2006**, *14*, 41–64. [[CrossRef](#)] [[PubMed](#)]

42. Dinges, P.; Agha, G. Targeted Test Input Generation Using Symbolic-concrete Backward Execution. In Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering, ASE '14, New York, NY, USA, 15–19 September 2014; pp. 31–36.
43. Bohme, M.; Paul, S. A Probabilistic Analysis of the Efficiency of Automated Software Testing. *IEEE Trans. Softw. Eng.* **2016**, *42*, 345–360. [[CrossRef](#)]
44. Liang, H.; Zhang, Y.; Yu, Y.; Xie, Z.; Jiang, L. Sequence coverage directed greybox fuzzing. In Proceedings of the 27th International Conference on Program Comprehension, ICPC 2019, Montreal, QC, Canada, 25–31 May 2019; pp. 249–259.
45. Zong, P.; Lv, T.; Wang, D.; Deng, Z.; Liang, R.; Chen, K. FuzzGuard: Filtering out Unreachable Inputs in Directed Grey-box Fuzzing through Deep Learning. In Proceedings of the 29th USENIX Security Symposium, Boston, MA, USA, 12–14 August 2020.
46. Serebryany, K.; Bruening, D.; Potapenko, A.; Vyukov, D. Addresssanitizer: A fast address sanity checker. In Proceedings of the USENIX Annual Technical Conference, Boston, MA, USA, 13–15 June 2012.
47. CVE-2018-11237. Available online: <https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-11237> (accessed on 30 May 2020).
48. Rustamov, F.; Choi, M.; Yun, J. Search-based Concolic Execution for SW Vulnerability Discovery. *IEICE Trans. Inf. Syst.* **2018**, *101*, 2526–2529.
49. Yan, S.; Ruoyu, W.; Christopher, S.; Nick, S.; Mario, P.; Andrew D.; John, G.; Siji, F.; Christopher, H.; Christopher, K.; et al. (State of) The Art of War: Offensive Techniques in Binary Analysis. In Proceedings of the IEEE Symposium on Security and Privacy, San Jose, CA, USA, 23–25 May 2016.
50. Deng, Y.; Chen, Y.; Zhang, Y.; Mahadevan, S. Fuzzy Dijkstra algorithm for shortest path problem under uncertain environment. *Appl. Soft Comput.* **2012**, *12*, 1231–1237. [[CrossRef](#)]
51. Darpa Cyber Grand Challenge. Available online: <http://archive.darpa.mil/cybergrandchallenge/> (accessed on 30 May 2020)
52. Fioraldi, A.; Maier, D.; Eißfeldt, H.; Heuse, M. AFL++: Combining incremental steps of fuzzing research. In Proceedings of the 14th USENIX Workshop on Offensive Technologies (WOOT 20), USENIX Security '20, Boston, MA, USA, 11 August 2020.
53. LibFuzzer—A Library for Coverage-Guided Fuzz Testing. Available online: <https://llvm.org/docs/LibFuzzer.html> (accessed on 30 April 2020).
54. American Fuzzy Lop Plus Plus (afl++). Available online: <https://github.com/AFLplusplus/AFLplusplus> (accessed on 30 March 2020).
55. Cadar, C.; Sen, K. Symbolic execution for software testing: Three decades later. *Commun. ACM* **2013**, *56*, 82–90. [[CrossRef](#)]
56. Libksba Source Code. Available online: <https://fossies.org/linux/privat/libksba-1.3.5.tar.gz/> (accessed on 29 April 2020).
57. Libjpeg Source Code. Available online: <https://www.ijg.org/files> (accessed on 20 April 2020).
58. Libarchive Source Code. Available online: <https://libarchive.org/downloads/> (accessed on 20 April 2020).
59. Tcpdump Source Code. Available online: <http://www.tcpdump.org/release/> (accessed on 21 April 2020).
60. Poppler Source Code. Available online: <https://poppler.freedesktop.org/releases.html> (accessed on 19 April 2020).