future internet

MDPI

Article

DeepDiver: Diving into Abysmal Depth of the Binary for Hunting Deeply Hidden Software Vulnerabilities

Fayozbek Rustamov †, Juhwan Kim and Joobeom Yun *

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.)

* Correspondence: jbyun@sejong.ac.kr

† Current address: Rm. #727, Daeyang AI center, 209 Neungdong-ro, Gwangjin-gu, Seoul 05006, Korea.

Received: 30 March 2020; Accepted: 16 April 2020; Published: 18 April 2020

Abstract: Fuzz testing is a simple automated software testing approach that discovers software vulnerabilities at a high level of performance by using randomly generated seeds. However, it is restrained by coverage and thus, there are chances of finding bugs entrenched in the deep execution paths of the program. To eliminate these limitations in mutational fuzzers, patching-based fuzzers and hybrid fuzzers have been proposed as groundbreaking advancements which combine two software testing approaches. Despite those methods having demonstrated high performance across different benchmarks such as DARPA CGC programs, they still present deficiencies in their ability to analyze deeper code branches and in bypassing the *roadblocks checks* (magic bytes, checksums) in real-world programs. In this research, we design DeepDiver, a novel transformational hybrid fuzzing tool that explores deeply hidden software vulnerabilities. Our approach tackles limitations exhibited by existing hybrid fuzzing frameworks, by negating roadblock checks (RC) in the program. By negating the RCs, the hybrid fuzzer can explore new execution paths to trigger bugs that are hidden in the abysmal depths of the binary. We combine AFL++ and concolic execution engine and leveraged the *trace analyzer approach* to construct the tree for each input to detect RCs. To demonstrate the efficiency of DeepDiver, we tested it with the LAVA-M dataset and eight large real-world programs. Overall, DeepDiver outperformed existing software testing tools, including the patching-based fuzzer and state-of-the-art hybrid fuzzing techniques. On average, DeepDiver discovered vulnerabilities 32.2% and 41.6% faster than QSYM and AFLFast respectively, and it accomplished in-depth code coverage.

Keywords: software vulnerability; hybrid fuzzing; concolic execution; patching-based fuzzing

1. Introduction

Software vulnerability is considered one of the foremost critical threats to any computer network and programs will inevitably have defects [1]. Many of these susceptibilities have the potential to exploit programs, often with malicious intent. Such imperfect codes pose critical threats to information security [2]. Therefore, it is essential to detect vulnerabilities that exist within components of a computer program.

Generally, it is accepted in the cybersecurity sphere that automated software analyzing approaches have achieved magnificent progress in discovering software vulnerabilities. In particular, fuzzing techniques have been applied [3,4] to discover popular program vulnerabilities [5,6], for instance, *"Month of Kernel Bugs"* [7], *"Month of Browser Bugs"* [8] and *"Heartbleed"* [9,10] bugs.

Fuzzing is the most powerful automated testing technique that discovers security-critical vulnerabilities and security loopholes in any program cost-effectively and rapidly by providing invalid or random data that is generated and feeding them to the program [11–13]. It has advanced into

a straightforward and efficient tool for the verification of code security and improvement of reliability. Moreover, fuzzing tools are widely used for various purposes. For instance, it is applied in *Quality Assurance (QA)* to analyze and secure internally developed programs, in *Vulnerability Assessment (VA)* to test and attempt to crack software package or system, and in *System Administration (SA)* to examine and secure a program in its usage environment.

Despite the abovementioned competencies of fuzzing, it still presents several weaknesses. The technique alone cannot deal with all security threats. To perform much more effectively in discovering serious security threats, it will require a significant amount of time [14]. It also has small-scale capabilities in achieving high code coverage.

To overcome the critical limitations of fuzzing, a hybrid approach called *hybrid fuzzing* [15] has been proposed recently, and after showing high-quality performance across various synthetic benchmarks, it has become increasingly popular in bug detection [16–18]. Research has shown that fuzzing and concolic execution [19] are powerful approaches in software vulnerability discovery; combining them can possibly leverage their remarkable strengths and possibly mitigate weaknesses in the program.

The cardinal idea behind hybrid fuzzing is to apply the fuzzing technique in exploring paths and taking strategic advantage of concolic execution for the purpose of providing an excellent solution to path conditions. More specifically, fuzz testing is efficient in exploring paths that involve general branches, not to mention its ability to test programs quickly. However, it is not effective in exploring paths containing specific branches that include magic bytes, nested checksums. In comparison, concolic execution [19–21] extensively applied in detecting a significant number of bugs, can easily generate concrete test cases, but still has a path explosion problem. For example, Driller [18] provides a state-of-the-art hybrid fuzzing approach. It demonstrated how efficient the hybrid testing system was in DARPA's Cyber Grand Challenge binary tests, discovering six new vulnerabilities that could not be detected using either the fuzzing technique or concolic execution. In addition, as generally understood, the hybrid fuzzing approach is limited due to its slow concolic testing procedure. Hence, QSYM [22] is tailored for hybrid testing that implements a fast concolic execution to be able to detect vulnerabilities from real-world programs.

Patching-based fuzzers can address the issue of fuzzing approaches from a different angle. Although this method was introduced to science a decade ago, it has not only shown remarkable results but has also been commonly applied in software vulnerability detection. To give an example, T-Fuzz [23] has recently been proposed and has shown efficiency in achieving of results. In short, it detects non-critical checks (NCC) in the target program and removes them to explore new paths and analyze hard-to-reach vulnerabilities.

Unfortunately, these hybrid testing and patching-based fuzzing approaches still suffer from scaling to detect vulnerabilities that are hidden in deep execution paths of real-world applications. To be more specific, studies have shown that [24], the coverage of functions located within the abysmal program depths are quite hard to reach. Consequently, the hybrid testing approaches discussed here fail to detect a significant number of bugs entrenched within the software's depths, and patching-based fuzzers such as the T-Fuzz [23] will have a transformational explosion problem when the true bug is hidden deeply in the binary. To tackle these issues, we design and implement DeepDiver, a novel transformational hybrid fuzzing method. We expect to increase the code coverage as well as bug detection capabilities of the hybrid fuzzer by negating roadblock checks in target programs.

Without a doubt, disabling certain roadblock checks may break the original software logic, and vulnerabilities detected in the negated RC program might include *false positives* [25]. To filter out false positives, we implement a *bug validator* based on the post-processing symbolic execution that enables us replicate true bugs contained in the original software.

To demonstrate how efficiently the new transformational hybrid testing approach performs in comparison to the most modern methods, we evaluated DeepDiver on LAVA-M [26] dataset programs that displayed vulnerability, as well as on eight real-world software applications.

DeepDiver discovered vulnerabilities in LAVA-M dataset in three hours whereas QSYM detected threats in five hours. Moreover, our tool outperformed all existing fuzzers like T-Fuzz [23] and AFLFast [27] in the presence of hard input checks and achieved significantly better code coverage.

The essential contributions of this paper are summarized as follows:

- We propose a set of novel approaches to aimed at improving the performance of our hybrid testing system. These approaches detect the fuzzing roadblock check and transform the target binary, including (a) analyzing checks of the target program in order to find a roadblock check that causes stoppage of fuzzing techniques, and (b) negating detected roadblock checks.
- To enhance hybrid fuzzing, we design DeepDiver with a bug validator. Bug validator determines whether a detected code weakness is a true bug or not and certifies the security strength by filtering false positives.
- We propose splitting floating-point comparisons into a series of the sign, exponent and mantissa comparisons for improving hybrid fuzzer capability if floating-point instructions are in the target program.
- We demonstrate the effectiveness of DeepDiver by testing LAVA-M dataset and eight real-world software applications. Comprehensive evaluation results show that our implementation can scale to various sets of real-world programs.

Accordingly, this research will proceed as follows. Section 2 annotates the motivation of our research. Section 3 and Section 4 depict the methodology and hybrid system design of DeepDiver in detail, respectively. Section 5 analyzes implementation and evaluation of the core techniques of DeepDiver where the security-related program flaws are found successfully in a short time. Section 6 introduces related work and explains the limitations of DeepDiver. Finally, we conclude this research paper in Section 7.

2. Motivation

One of the most efficient solutions for increasing code coverage is the coverage-guided fuzzing approach. Some state-of-art-coverage-guided fuzzing techniques including AFL [3], Hongfuzz [28], LibFuzzer [29] focus on the increasing ability of code coverage and quickly discover software-application flaws. To trigger new code paths, fuzzing tools focus on the mutating seed inputs while generating new test cases.

However, fuzz testing methods have a generally known drawback. Figure 1 illustrates the challenges faced by the fuzzer in discovering bugs hidden in the deep parts of target programs. The structure of software applications is complex, especially if it contains several magic bytes and nested complex checks (say, checksums). This type of check has always made the software vulnerability testing system difficult. Checksums [30,31] are a typical way of analyzing the probity of data widely used in real-world software applications, in network protocols (for example, TCP/IP) and in various file formats (such as ZLIB, PNG). It is too difficult for software testing approaches to generate valid input to bypass checksums as well as magic bytes, and the path of the fuzzer can be blocked. Specifically, mutational fuzzing tools are incapable of passing through complex roadblock checks, and when programs have checksum instruments, there is a high probability of the rejection of generated seeds in the early stage of running the program. As a result, if the path is hard to explore, fuzzers get 'stuck' and choose other paths that are easier to take without covering the new path.

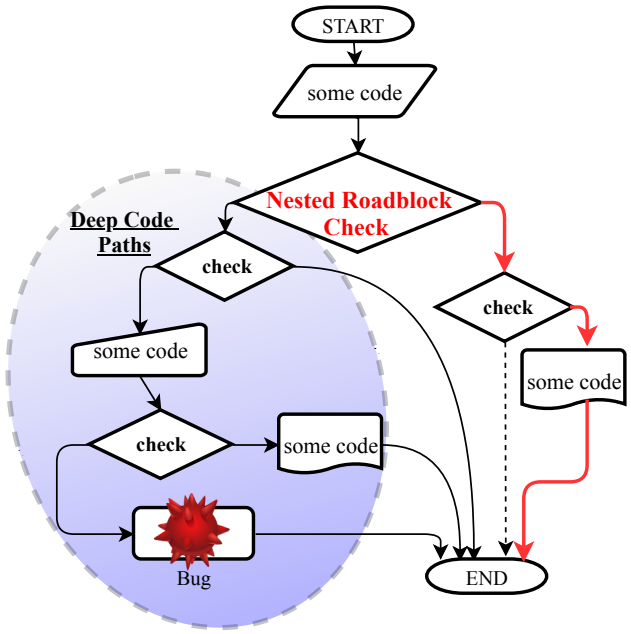

Figure 1. Challenge for fuzz testing methods.

Listing 1 illustrates a simplified scenario and issues regarding existing fuzzing approaches that explain our motivation as well as common roadblock checks. In this example, there are various complex roadblock checks and magic bytes. It is widely known that passing some nested checks for generated inputs is not straightforward.

Listing 1. Motivation example.

```
#define BUF_SIZE 100
#define SIZE 5
int main(int argc, char **argv){
    unsigned char input[BUF_SIZE];
    char source[SIZE];
    gets(source);
    // magic byte example
    if (strcmp(source, "HELLO") == 0){
        printf("Matched succesfully! \n");
    }
    else {
        EXIT_ERROR("Inappropriate data \n");
    }

    // nested complex sanity check example
    if (input[0] == 0xAB && input[1] == 0xCD){
        printf("Pass the next check succesfully! \n");
        if (input[16] == '*' && input[17] == 'P'){
            // trigger the BUG
        }
        else {
            // some common codes without vulnerabilities
        }
    }
    else {
        ERROR("Inappropriate bytes \n");
        // some other codes
    }
    return 0;
}
```

For this motivating example, the majority of modern coverage-based fuzzers and hybrid testing systems have many limitations and penetration power restrictions to pass through the path blocked by magic bytes comparisons. For instance, the string *"HELLO"* in line 8 of the program is five magic bytes. To bypass line 8, the state-of-the-art fuzzers must mutate all the five magic bytes all at once and do so correctly. The reason is that mutating *"HAAAA"*, *"HEAAA"*, *"HELAA"*, *"HELLA"* correctly cannot lead to a new execution path. Under these circumstances, it is too complex for coverage-based fuzzers to cover the magic bytes, not to mention the numerous executions that need to be investigated to pass magic bytes. The challenge is that the most modern coverage-based fuzzing tools are incapable of knowing the location of magic bytes that are present within the input of the test, causing these approaches to waste time trying to match magic bytes successfully.

Another common challenge for fuzz testing techniques is effectively fuzzing checksums. It is very difficult to generate accurate inputs to pass complex checksums for modern constraint solvers. A typical sample of nested condition statements is depicted from line 16 to line 18 in Listing 1. To reach line 18, which triggers the bug, the first input must satisfy the constraint at line 16. Secondly, after passing the first check successfully, it must satisfy the next check at line 18.

Recently, several advanced fuzzing techniques have been made to address the above challenge. One example is the symbolic analysis-based hybrid testing methods such as Driller [18]. To analyze deeper execution paths, Driller combines concolic execution and AFL. This technique can learn magic bytes. In the given motivating example, Driller can solve magic byte comparison quickly, and easily pass line 8. However, it is incapable of exploring nested complex roadblock checks, and suffers from a path explosion problem.

Moreover, state-of-the-art fuzzer VUzzer [32] uses dynamic taint analysis to explore those magic byte and nested checks and emphasizes to the generation of test cases which can execute deeper paths. However, the execution speed is disappointingly slow owing to the dynamic taint analysis of VUzzer.

Another method is the AFL-lafintel [33]. To convert magic bytes, the AFL-lafintel uses program transformation and divides magic bytes into small portions. However, it has a path explosion issue that prevents it from analyzing the location of magic bytes.

Furthermore, one of the most efficient patching-based fuzzing techniques, the T-Fuzz [23], has indicated high performance in discovering software vulnerabilities. By using the dynamic tracing method, it identifies non-critical checks and then flips the condition of the conditional jump instruction to explore new execution paths. NCC is a sanity check that separates orthogonal data and removing NCC enables the fuzzer to explore new paths [23]. However, one of the major limitations of T-Fuzz is that when a vulnerability is located in depths of a path under the many non-critical checks, the T-Fuzz transforms the binary too many times and causes problems with hard disk memory overuse.

3. Methodology

DeepDiver is designed to find and analyze the roadblock checks of the target program, and then patch these checks to execute new paths. By doing this, we will able to analyze the abysmal depth of the binary and can overcome the limitations of T-Fuzz [23] by leveraging the concolic execution engine. Analyzing the process of hard-to-reach codes consists of two main steps in our transformational hybrid fuzzer. The first is detecting the roadblock checks that include nested complex checks (like checksums) and negating them to execute the new paths. The second step is exploring those new paths with the hybrid fuzzing approach.

Detection and negation process of roadblock checks. To identify the roadblock checks, we first collect the coverage statistics that include branch and edge coverage information. The RCs are found through the trace analyzer approach that constructs the tree for each input and analyzes the coverage statistic details of the target. The next stage is the negation of the detected RCs, and there are various methods for handling this. For example, the Steelix [16] leverages Dynamic Binary Instrumentation (DBI) to split nested complex checks into smaller ones, and the Static Binary Translation (SBT) is often used to change the control flow graph (CFG) structure.

However, we decided to flip the position of conditional jump instruction which is uncomplicated and requires light work. Figure 2 highlights the CFG in Listing 1 and the process followed after flipping the detected roadblock check conditions. The nested roadblock check in Listing 1 include *S2*, *S3*, *S4*, and *S5* checks. When mutation-based fuzzers and concolic execution engines are used alone, bypassing these nested complex checks will be too difficult. Those approaches must generate input that satisfies all *S2*, *S3*, *S4*, and *S5* checks, otherwise, the bugs hidden in the deep parts stay without getting triggered. After detecting that roadblock check and flipping the conditional jump instruction's condition, fuzzing technique can trigger the vulnerability easily. A detailed explanation concerning detection and negation processes is given in Section 4.2.

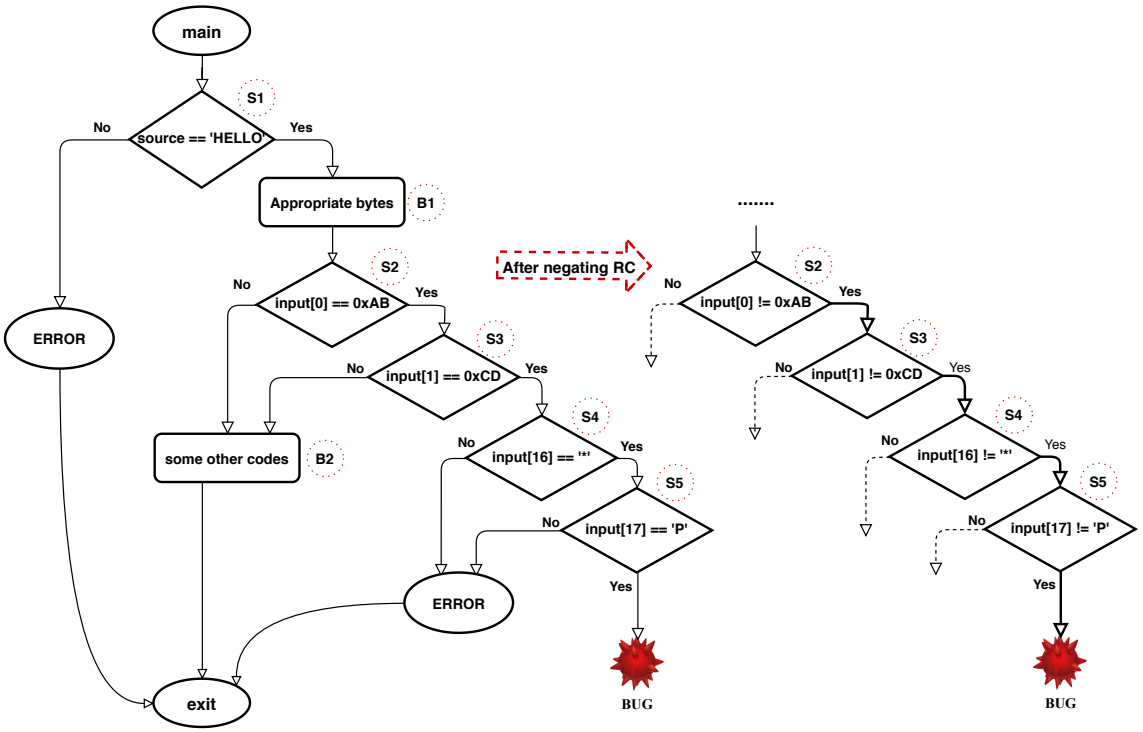

Figure 2. CFG of Listing 1 and after transforming process.

Hybrid fuzzing technique. In this approach, a combination of AFL++ [34] and concolic execution engine allow us to dive deeper and deeper into the binary. We choose AFL++ [34] for DeepDiver because it integrated many high qualified software vulnerability testing tools. Although concolic execution is slower than fuzzing, we leverage it only for tackling the difficult path conditions after bypassing the complex roadblock checks. Additionally, following our evaluation of the T-Fuzz in LAVA-M dataset, it showed significant limitations of this approach. When we tested each binary of LAVA-M dataset three times, due to the program transformation process, it filled a 50GB HDD in 24 h. In our findings, this limitation of the T-fuzz can fill a 1 TB HDD easily. To overcome this challenge, the concolic execution engine is vital in exploring the hardest path conditions. Thus, the concolic execution's ability has been widely applied in the DeepDiver approach. Basically, the concolic execution engine takes a test case from the seed pool of AFL++ and produces new inputs that can execute new paths. Detailed information is given in Section 4.3.

Bug Validation. The last step is the Bug Validation phase, which verifies whether the crash inputs found are true or false. In this step, we leveraged a pre-constrained tracing tool with a transformation-aware approach. This mode gathers path constraints and provides true bugs by filtering out the crash inputs.

This research is not the first to use the hybrid fuzzing approach to discover crashes in software applications, but to the best of our knowledge, DeepDiver is the first research that investigates

transformational hybrid fuzzing. This implementation not only improves the capability of faster coverage of vulnerabilities but will also allow the analysis of vulnerable functions that are set deeper in programs.

4. System Design

In this section, we describe the workflow design and the main components of DeepDiver in detail. First, we explain the architecture of our hybrid fuzzing approach based on the binary patching in Section 4.1. Secondly, we discuss an approach that detects nested complex roadblock checks in Section 4.2. Next, we present a hybrid fuzzing technique including the most modern, fast and powerful software testing approaches: (a) the AFL++ [34] fuzzing tool and splitting floating-point instructions, and (b) the fast concolic execution in Section 4.3. Finally, we introduce the bug validator method that ensures the crash inputs that are found can cause the original program to crash as explained in Section 4.4 below.

4.1. Overview of the Deep Diver Architecture

To explain the absolute efficiency of our implementation, we demonstrate the architecture of DeepDiver in Figure 3, which comprises several models: the complex roadblock check detector and patching binary, the hybrid fuzzing approach including AFL++ 2.60d [34], and the concolic execution, a bug validator.

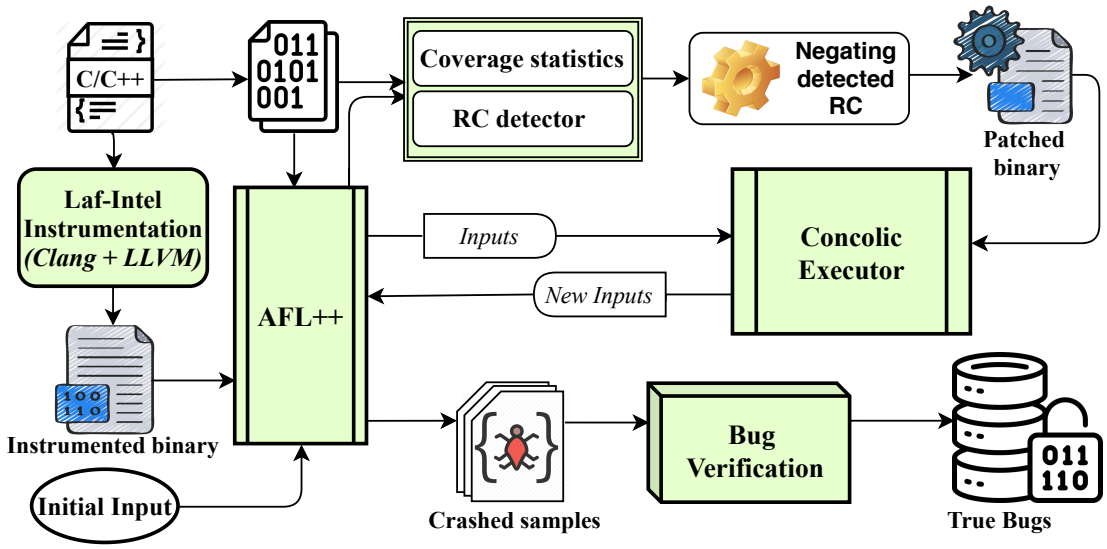

Figure 3. Overall workflow of the DeepDiver.

The fundamental step of DeepDiver's function is to detect roadblock checks from the target, which is the non-instrumented binary and negating the detected RCs automatically. RCs are some magic values and checksums which are not intended to prevent bugs. Negating RCs do not incur erroneous vulnerabilities. In addition, by the process of detection and negation of RCs, the fuzzer can be accessible to explore the newly blocked execution paths and the potential code defects can be discovered easily. However, there are critical roadblock checks that are crucial to the program's functionality. It will break the logic of software if we negate the CRC.

The next step is the analysis of the target programs by the hybrid fuzzing approach including AFL++ [34] and concolic execution. We instrument the target program for AFL++ [34] fuzzer by LLVM Laf–Intel Instrumentation and *afl-clang/afl-clang-fast/afl-clang-fast++*. Concolic execution analyzes patched binary whose RCs are negated. We provide a detailed description of our hybrid fuzzer and an explanation why we decided to use AFL++ rather than other fuzzing tools (AFL, AFLFast for example) in Section 4.3.

The last step of our implementation is bug validator. It can determine whether the crash input that is found is a true bug or not by filtering out false positives. It helps to identify true crash inputs that can trigger the bug in the original program.

4.2. Locating RC Points and Negating Procedure

In this phase, DeepDiver uses the RC detector that pinpoints the locations of roadblock checkpoints within the target binary. There are various methods of locating the RC point in the program. One of them is TaintScope [35]. This is the first checksum-aware directed fuzz testing technique that can analyze checksum fields in input instances, accurately locate the RC in the target program according to branch profiling as well as concrete and dynamic taint tracking, and bypass detected RC points by execution flow alteration. Another option is using control flow analysis and complex data flow to track cohesion between the complex conditional jump and input. Despite the obviously impressive performance, those approaches are very high overhead and relatively costly (time-consuming).

Conditional jump instructions entail a compilation of checks. It is depicted as a node in the program as well as two edges coming out from node which can either be False or True [23] in the CFG. If generated input cannot bypass the conditional jump, it passes either the False or True edge. The key idea behind this step is to find conditional jump instructions that are not covered by fuzzer-generated inputs and gather them for negation. The process of detecting RC candidates consists of two main steps: *Coverage Information* and *RC detector models*.

To locate RC points, DeepDiver works as follows. First, DeepDiver starts fuzzing non-instrumented target binary with an initial input and during the fuzzing process if nodes and edges covered under the input, statistics of node coverages and edge coverages are collected by coverage statistics model. Then, if generated inputs cannot bypass the check, the detector designates that check as a RC candidate. The next step identifies many RC candidates by applying the *trace analysis* technique. After the detection process, information about all RC candidates' locations are recorded. Lastly, detected RC candidate locations will negate. To do this, we use *radare2* [36] automatic reverse-engineering framework python script named *r2pipe* [37,38]. It should be noted that detected RC candidates may incur false positives.

We demonstrate Algorithm 1 to explain the *Coverage Information* process. In this process, the algorithm uses three inputs including *TB* (the non-instrumented binary) as a target, the AFL++ *FuzzingTool* and initial input, *Seed*. This process gives coverage information *CovInfo* as output. In line 4, to generate new seeds, the *FuzzingTool* analyzes target binary TB with the initial *Seed*. Next, coverage data is collected for each basic block *BB* analyzed under each new seed in the *SeedPool*. All covered destinations stored into *CovInfo* in line 7–9, as well as generated *NewSeeds*, are gathered in the *SeedPool*. By this procedure, we can easily distinguish covered and uncovered destinations, and as a result, it significantly contributes to the detection of RC locations.

Algorithm 1 Coverage information

INPUT: *TB* (Target Program)
INPUT: *FuzzingTool* (AFL++)
INPUT: *Seed* (Initial input)

```
1:  CovInfo ←Ø
2:  SeedPool (NewSeeds) ←Ø
3:  while True do {
4:      SeedPool ← FuzzingTool(TB, Seed)
5:      for seed ∈ SeedPool do {
6:          Coverage ← GainedCoverage (TB, Seed)
7:          for BB ∈ Coverage do {
8:              Mark ← Info (BB)
9:              CovInfo ← CovInfo ∪ Mark
10:         }
11:     }
12:     SeedPool ← SeedPool ∪ NewSeeds
13: }
```

OUTPUT: Coverage information *(CovInfo);*

Algorithm 2 shows the detection process of RC destinations, and it uses three inputs. These include the coverage information *CovInfo* as the output of Algorithm 1, *SeedPool* which has gathered new inputs from the fuzzer and CFG control flow graph of the target program. The key points of Algorithm 2 are to first construct the tree for each input, then take a *branch* from *TreeConstr*, and then get that branch neighbor which is located below from that *branch* and finally check that *Neighbor* is covered by the fuzzing tool in Algorithm 1. If the *Neighbor* of the branch is not located in the covered information *CovInfo*, that *branch* is considered the RC destination. In addition, *TreeConstr* is built by merging *traces* which are taken by trace analysis for each *seed* in the *SeedPool* and branch neighbors are *produced* by analyzing the target program CFG.

Algorithm 2 RC locations

INPUT: *CovInfo* (Coverage information)
INPUT: *SeedPool* (New seeds taken from AFL++)
INPUT: *CFG* (Target binary)

```
1:  RC locations ←Ø
2:  TreeConstr ←Ø
3:  for seed ∈ SeedPool do {
4:      trace ← TraceAnalyzing(seed)
5:      if trace ∉ TreeConstr then {
6:          TreeConstr ← TreeConstr ∪ trace
7:      }
8:  }
9:  for branch ∈ TreeConstr do {
10:     branchNeigh ← PickUpNeighbor(branch, CFG)
11:     Branches(branchNeigh)++
12:     for Neighbor ∈ Branches do {
13:         if Neighbor ∉ CovInfo then {
14:             RC ← RC ∪ branch
15:         }
16:     }
17: }
```

OUTPUT: RC candidate locations;

The negation process of RC destinations is straightforward. After RC detection process condition of the RC conditional jump instruction is flipped by DeepDiver. To achieve this, we leveraged the *r2pipe* [37,38] python script.

4.3. Hybrid Fuzzer Approach

Fuzz testing approach. Although there are efficient fuzzing techniques such as AFL [3], AFLFast [27], Hongfuzz [28] and VUzzer [32] in this research, we decided to use AFL++ [34] fuzzing tool with concolic execution. The rationale is that it is a highly efficient fuzzing tool and has been integrated with a significant number of excellent software vulnerability testing tool features. For example, AFLFast's *Power Schedules* [39], improved the seed selection strategy of AFL. It also smartly arranged the generated inputs from a seed, MOpt [40] mutator, InsTrim [27,41] very efficient CFG LLVM mode instrumentation for real-world applications, Radamsa [42] mutator and others. These features make AFL++ a more powerful and effective fuzzing technique than other existing state-of-the-art fuzzers.

In addition, to extend vulnerability finding capability, DeepDiver supports floating-point instructions. We can see floating-point instructions in many real-world applications such as *ffmpeg* and there are some file formats that use raw floats in the specification. While instrumenting the target program by Laf-Intel instrumentation in LLVM mode, floating-point instructions can be split, and it creates opportunities for the fuzzer to analyze the target better. In addition, the example given in Listing 2 illustrates the next problem with the existing fuzzing approaches. Assume *magic number* takes *1.23* as an input in Listing 2 simple program. The fuzzer probability of generating an input that is exactly *1.23* to explore the vulnerable functions is very low. After the floating-point splitting process, the probability of the bypassing a comparison such as line 8 in can be higher since three chained partial comparisons are used. Quite specifically, the floating-point instruction split process divides the comparison into small comparisons splitting *mantissa, sign* and *exponent* and splitting the *mantissa* and *exponent* again into bytes comparisons. The probability of the triggered application flaws detection can be high, first if the sign is right, then if the exponent is right, and finally if the mantissa part is right. Also, exponent and mantissa comparisons are split into chained 8-bit comparisons. That means there are many more partial steps with corresponding instrumented basic blocks, which allow AFL++ to monitor progress better when trying to find the right floating-point value.

Listing 2. Small floating -point example.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(void)
5  {
6      float magic_number, x;
7      scanf(%f, &magic_number);
8      if (x == magic_number) {
9          // trigger the BUG
10     }
11     return 0;
12 }
```

Concolic execution approach. Concolic testing is extensively used in different security-related testing approaches, such as automatic test input generation [43,44]. In this research, the purpose of applying the concolic engine is to effectively generate new test cases which can cause the exploration of new execution paths. In addition, discovering hidden bugs which lay in the depths of target programs are easy and fast to find with the concolic execution technique.

The concolic testing process receives program inputs and generates a set of new concrete inputs; it runs the patched binary on both symbolic and concrete values. At the beginning of the concolic execution step, it gets a seed from the fuzzer and patched binary, which are negated RCs. After that, it starts to generate new concrete inputs which can cover new execution paths. To be exact, when it reaches the nested complex check location, concolic testing engine generates the concrete inputs and sends that inputs back to the AFL++ fuzzer. As a result, the fuzzing technique can bypass the complex roadblock check branches easily.

A concolic execution engine of the DeepDiver ported from QSYM [22] and Dynamic Binary Translation (DBT) [22,45] used in order to natively execute the patched binary and choose basic blocks to symbolic execution. Newly generated concrete test cases from the concolic execution engine help the fuzzer to dive deeper into the target binary, bypass the magic byte comparisons, as well as nested complex checks and extend code coverage.

4.4. Bug Validation

As mentioned above, the purpose of DeepDiver is to analyze hard-to-reach codes and make paths reachable to discover vulnerabilities hidden in abysmal depths of binary by negating RC destinations. If DeepDiver finds software vulnerabilities after the RC negation process, those vulnerabilities may incur false positives. By this, we mean that the crash input cannot cause the crash in the original program. Thus, we use a bug validation mode to ensure that crashing input in the patched binary can trigger the bug in the original program by reproducing true bugs.

In the bug validation mode, we use a pre-constrained tracing tool with a transformation-aware approach proposed by T-Fuzz [23]. In this validation phase, patched binary which detected RCs are negated. Locations of negated RCs, the crashing input which causes crashes in the patched binary, and the crashing addresses are used as input.

The bug validation mode gathers path constraints that can demonstrate whether the crashing input in the patched binary is false positive or not. The first crash input is transmuted to a pre-constrained tracing tool. By doing this, the trace can follow the code path. Next, to maintain track of the constraints in the patched binary and the original binary, bug validation mode traces the patched binary with the crashing input by the pre-constraint trace method. For one crash input, regardless of how many negated conditional jump instructions lie in the crash path in the patched binary, all that or path constraints added to the original program. After analyzing original binary constraints, if the constraint of the original program is satisfactory, that crashing input can cause a crash in the original target. If it is not satisfactory, the bug validation mode considers a constraint as a false positive.

5. Implementation and Evaluation

In this section, we introduce implementation and comprehensive evaluation to highlight the efficient performance of DeepDiver hybrid fuzzer based on the removal of roadblock checks in the binary. To determine the effectiveness of our hybrid fuzzing approach, we divide the evaluation into two extensively used benchmarking groups, including LAVA dataset [26,46] which inserted many different hard-to-reach vulnerability functions, and a second group consisting of eight real-world programs. Table 1 shows details about the first group, LAVA-M dataset.

Table 1. LAVA-M dataset statistics.

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

We demonstrate comparative results of our application by using effective fuzzing techniques. These include AFLFast [27] which provides effective power schedules, AFL [3] as a coverage-guided fuzzing tool, T-Fuzz [23] which is an efficient fuzzing approach based on program transformation strategy as well as other state-of-the-art hybrid testing techniques, such as DeepFuzz [47] and QSYM [22].

5.1. Implementation and Experiment Setup

We have implemented the DeepDiver hybrid fuzzing technique consisting three major modules, i.e., the RC detector, the hybrid fuzzing module based on AFL++ 2.60d [34] and concolic execution engine, the bug validation. The RC detector module was built using the angr tracer [48], and the detected RCs negation component was implemented using *radare2* [36] automatic reverse-engineering framework python script named *r2pipe* [37,38]. The concolic executor engine was implemented atop QSYM [22]. The bug validation module was built by angr [49].

We ran all experiments on a machine with 2.7 GHz Intel(R) Core (TM) i5-6400 processor and 32 GB of memory and we used Ubuntu 16.04 LTS version. We used three cores for AFL++ master, AFL++ slave and concolic execution engine, respectively.

5.2. LAVA Dataset Evaluation

The LAVA dataset involves many various vulnerable programs and is commonly used to evaluate software code defect finding tools. LAVA-M dataset is the part of the LAVA dataset comprising four buggy programs *base64*, *uniq*, *md5sum* and *who*. Evaluation time for each program was three and five hours, and we tested each binary five times and summarized the average results. The statistics of LAVA-M dataset target programs are illustrated in Table 1, including several edges, basic blocks, instructions, the total listed bugs and test options [50].

We have evaluated our implementation against two hybrid testing tools, DeepFuzz [47] and QSYM [22], in addition to another program transformation fuzzer T-Fuzz [23]. Unfortunately, we cannot test those programs on the Driller [18] hybrid fuzzer because LAVA-M dataset applications are not DECREE binaries. Table 2 shows evaluation results in the first group dataset, arrived at by testing techniques mentioned above. Moreover, we borrowed QSYM (P) column results from the QSYM paper and the run time was only five hours. According to the Qsym [22] paper, QSYM detected 100% of vulnerabilities in three binaries of LAVA-M dataset including *base64*, *uniq*, *md5sum* and 57.9% of bugs detected from *who* binary. QSYM showed high-performance results in the evaluation of LAVA-M dataset.. Therefore, we tested LAVA-M dataset by QSYM tool in our machine and QSYM (R) column represents results from our evaluation. According to the results from the three-hour run time, DeepDiver outperformed them, discovering 88.6% of *base64*, 92.8% of *uniq* and 84.2% of *md5sum* listed bugs in three hours. On average, DeepDiver discovers vulnerabilities 37.6%, 48.5% and 28.8% faster than QSYM, DeepFuzz and T-fuzz, respectively. Our result in the testing binary named *who* is 101 bugs, and it is not even comparable with the QSYM paper's original result which recorded a large number of bugs—1238 in five hours. However, when we evaluated the QSYM with optimistic solving on the LAVA-M dataset five times in our machine, the average number of bugs detected was 71. Probably, one of the reasons why the test results did not match was because our machine was slower than QSYM authors' server machine, which was Intel Xeon E7-4820 8 2.0 GHz cores with 256 GB of RAM [22].

Table 2. The number of found bugs in testing on LAVA-M dataset.

Program	Listed Bugs	T-Fuzz		DeepFuzz		DeepDiver		QSYM (R)		QSYM (P)
		3 h	5 h	3 h	5 h	3 h	5 h	3 h	5 h	5 h
base64	44	21 47.7%	43 97.7%	15 34.0%	29 64.9%	39 88.6%	44 100%	22 50%	38 86.3%	44 100%
uniq	28	16 57.1%	26 92.8%	2 7.14%	16 57.1%	26 92.8%	28 100%	6 21.4%	13 46.4%	28 100%
md5sum	57	27 47.3%	49 85.9%	19 33.3%	30 52.6%	48 84.2%	57 100%	26 45.6%	44 77.1%	57 100%
who	2136	41 1.91%	63 2.94%	12 0.56%	21 0.98%	79 3.69%	101 4.72%	34 1.59%	71 3.32%	1238 57.9%

Table 3 illustrates the number of discovered crashes filtered out by Bug Validation and split floating-point instruction numbers in LAVA-M dataset. According to these results, after filtering out reported crashes the number of false positives was very low, indicating the efficient performance of Bug Validation mode.

Table 3. Filtering out detected crashes by Bug Validation and the number of split floating-point instructions in LAVA-M dataset.

Program	Reported Crashes	True Bugs	False Bugs	Split FP Instructions
base64	51	44	7	41
uniq	36	28	8	33
md5sum	63	57	6	72
who	111	101	10	159

Figure 4 shows the number of explored paths in the LAVA-M dataset. We compared DeepDiver coverage results which were a fuzzing execution time of three hours with the state-of-the-art hybrid fuzzers QSYM [22] and DeepFuzz [47]. Results show that during the first hour of the execution time results of all hybrid fuzzing techniques were almost the same in *base64, md5sum* and *who* applications. When analyzing the *uniq* binary, the coverage results of QSYM were slightly higher in the first hour than DeepDiver. All hybrid testing approaches showed nearly the same coverage results in the middle of the all LAVA-M software analysis process. However, at the end of the testing process, DeepDiver reached the highest result compared with both hybrid fuzzing tools except binary *who* where path coverage results of DeepDiver were slightly higher than QSYM. Despite the execution time has been three hours, DeepDiver explored the many paths. Without a doubt, AFL ++ power schedules integrated from AFLFast [27] and concolic execution play a vital role in covering many paths.

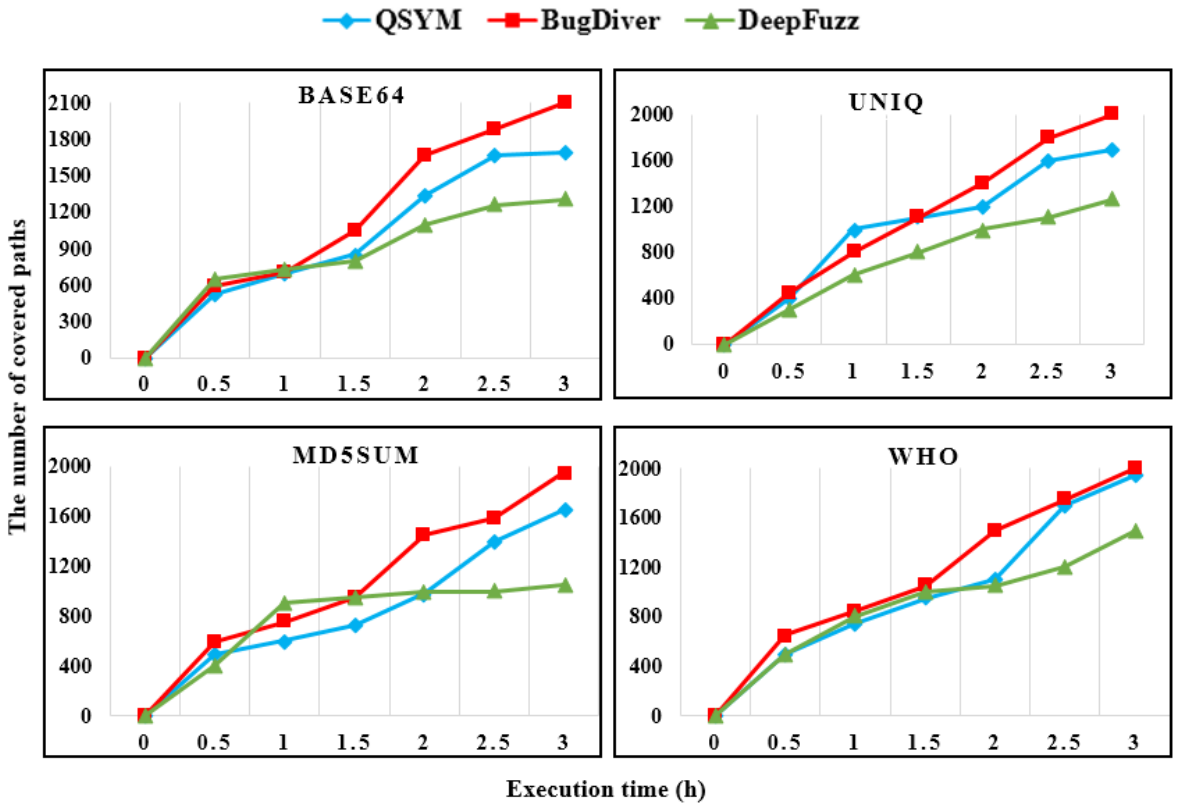

Figure 4. Number of covered paths in the LAVA-M dataset.

5.3. *Evaluation of Real-World Programs*

In this evaluation, we used 8 real-world non-trivial software applications that have been tested by other state-of-the-art fuzzing techniques for a long period of time. Table 4 shows detailed information regarding these applications. By fuzzing the second group programs, we compared DeepDiver's efficiency with the high-performance fuzzing tools, including AFL [3], AFLFast [27], and two hybrid fuzzers, QSYM [22] and DeepFuzz [47]. The run time for each program was six hours, and we analyzed each program five times to summarize the average results.

Table 5 illustrates the number of unique crashes found by evaluating real-world programs. According to the results, our implementation overcame other state-of-the-art fuzzers and hybrid testing techniques. A total of 159 unique crashes were detected in six hours by DeepDiver. When compared with hybrid fuzzers QSYM and DeepFuzz, 120 and 56 total unique crashes were found, respectively. AFLFast discovered 98 crashes, and the lowest result was recorded by AFL with 13 unique crashes.

Table 4. Statistics on the eight evaluated real-world software applications.

Package	Target Programs	Version	Class Type	Basic Blocks	Edges	Input Format	Test Options	Source
binutils	readelf	2.30	dev	21,249	31,086	binary	-a @@	[51]
binutils	objdump	2.30	dev	43,935	74,313	binary	-d @@	[51]
libksba	cert-basic	1.3.5	crypto	9958	14,120	crt	@@	[52]
libjpeg	djpeg	9c	image	4844	6776	jpg		[53]
libarchive	bsdtar	3.3.2	archive	31,379	43,390	tar	@@	[54]
tcpdump	tcpdump	4.9.2	net	33,743	48,791	pcap	-r/-nr @@	[55]
poppler	pdftohtml	0.22.5	doc	54,596	71,945	html/pdf	@@	[56]
poppler	pdftotext	0.22.5	doc	49,867	62,391	pdf/txt		[56]

Table 5. The number of detected unique crashes in 6 h.

Target Programs	AFL	AFL-Fast	QSYM	DeepFuzz	DeepDiver
readelf	8	27	25	14	48
objdump	0	30	31	12	35
cert-basic	1	12	11	6	13
djpeg	1	2	8	2	10
bsdtar	2	14	16	8	20
tcpdump	0	4	12	5	14
pdftohtml	0	0	3	0	3
pdftotext	1	9	14	9	16

Among these software vulnerability testing techniques, QSYM can compete with DeepDiver, but the number of detected unique crashes is low because QSYM uses AFL, and the efficiency of AFL is not as high as that of AFL++. We leveraged AFL, AFLFast and AFL++ for our implementation, and determined that the most suitable technique for DeepDiver was AFL++, which had integrated many efficient features from the other state-of-the-art fuzzing techniques. In addition, we measured the number of paths covered to highlight the efficient performance of DeepDiver, and Figure 5 demonstrates the comparison results with abovementioned fuzzers. In general, the DeepDiver has achieved higher explored path results than its other competitors. Despite the fact that the DeepDiver and QSYM reached the same results at the sixth hour of the test time, in analyzing the *pdftohtml* and *pdftotext* binaries, DeepDiver explored almost all paths in the middle of the test.

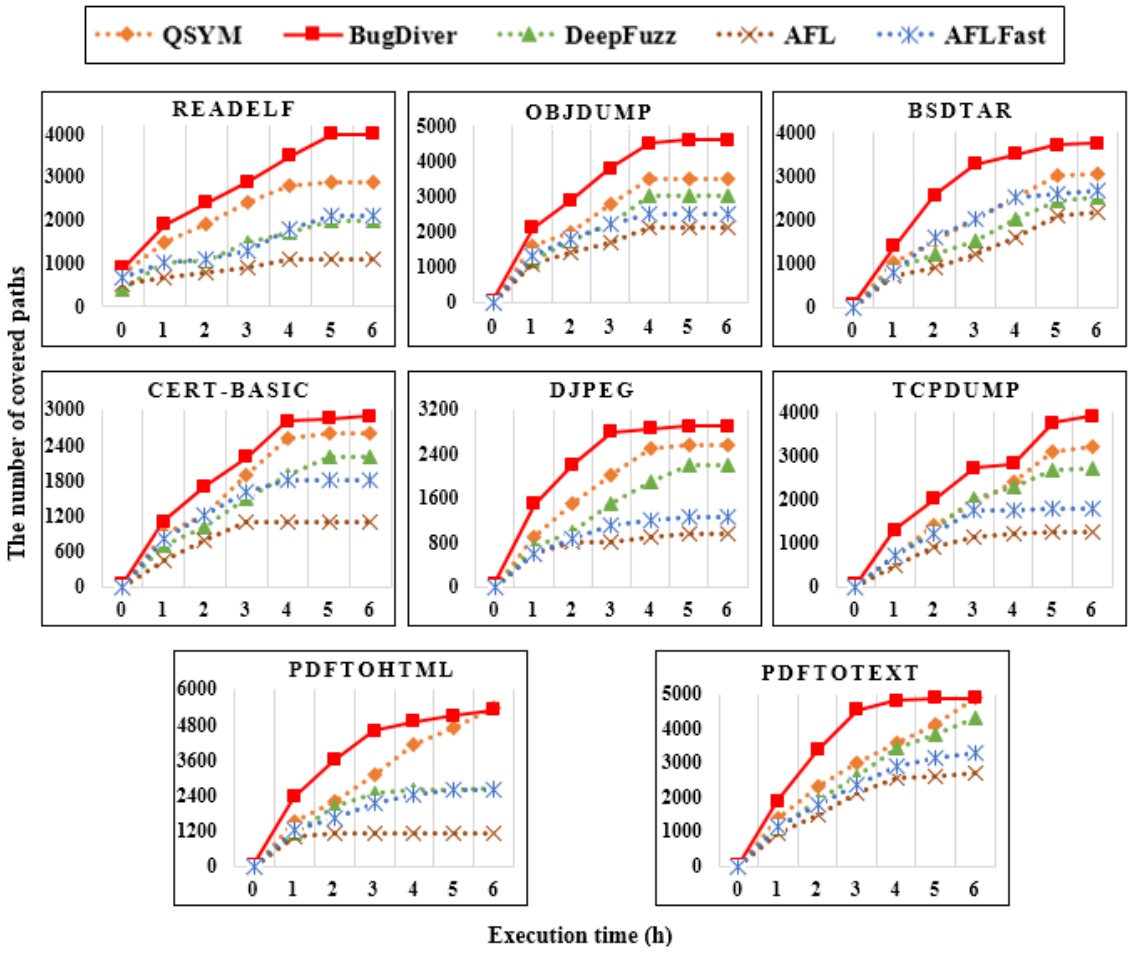

Figure 5. Number of covered paths in the real-world programs.

Figure 6 illustrates the evaluation results of execution speed per second on eight real-world programs. According to the bar chart, it is clear that the average number of executions of DeepDiver is the highest compared to the other competitors. When fuzzing the *djpeg* binary, the execution speed of DeepDiver, QSYM and AFLFast were almost the same, whereas in *tcpdump* binary, those fuzzing techniques results were slightly different. In addition, AFL fuzzing tool showed the lowest performance in our evaluation. The reason is that it is very difficult to bypass nested complex checksums for traditional fuzzer like AFL. In other words, AFL requires investigation of the numerous executions to match magic bytes in the target program and if AFL cannot generate seed to bypass magic bytes or nested checksums it becomes stuck. Moreover, the execution speed performance of DeepFuzz was slightly higher than AFL. DeepFuzz combines a dynamic symbolic execution with AFL to dive deep into the binary; however symbolic execution-based tools are not able to fuzz faster because of the path explosion issue. Additionally, symbolic testing tools suffer from the symbolic emulation problem, and it gets slow-down in formulating path constraints [22]. However, in our deduction, the execution speed performance of our implementation is the fastest, and it overcomes the other state-of-the-art fuzzing techniques. On average, DeepDiver detects software vulnerabilities 32.2% faster than QSYM and 41.6% faster than AFLFast. This is due to the fact that AFL++ uses the Power Schedules [27,39] feature, which has been integrated from AFLFast and other seed mutating features from excellent fuzzing tools. Furthermore, the execution performance speed of fuzzing tools is dependent upon the inputs in the seed pool. The value of execution speed per second drops if the seed pool is filled with slow inputs, but it cannot influence DeepDiver's fuzzing performance.

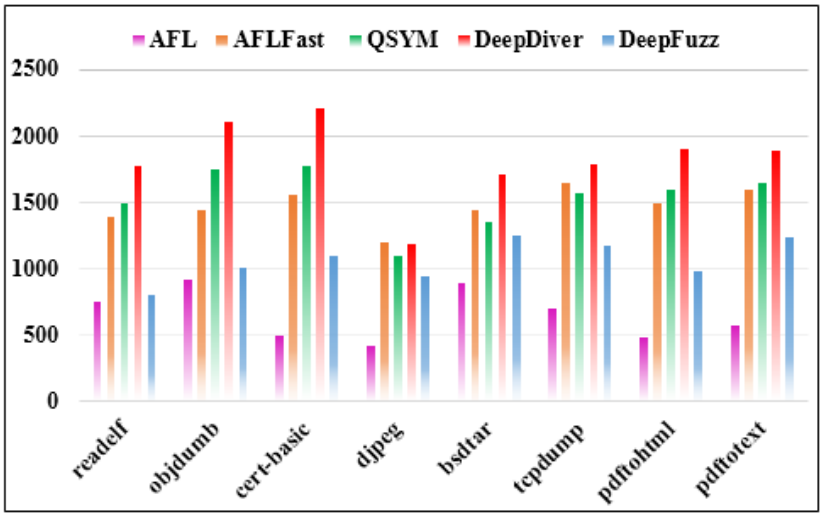

Figure 6. Execution speed evaluation on the real-world programs (from 0 to 2500 executions per second).

6. Related Work

Coverage-guided fuzzing. Coverage-guided fuzzing is an effective technique widely applied in software vulnerability detection. Since AFL [3] has demonstrated its efficient features in improving code coverage, the coverage-guided fuzzing approach became popular. Furthermore, CollAFL [50] performs accurate code coverage for mitigating path collisions, and AFLFast [27] highly improved the scheduling feature of AFL, thereby achieving huge code coverage in a short time.

However, it is very difficult for the coverage-guided fuzzing techniques to path branches that consist of magic bytes, nested complex checks, and checksums. To solve this issue, Steelix [16] bypasses the magic values branches easily by detecting comparison progress details against magic values. VUzzer [32] can easily cover magic bytes by application-aware mutation approach. If bugs are protected by nested complex checks, neither of these techniques can bypass in real-world programs.

Patching-based fuzzing. Symbolic and concolic execution-based tools can generate inputs that can cover the nested complex checks (such as checksums). However, these software testing tools are not able to fuzz faster after the complex checks; eventually, they cannot dive deeply to explore hard-to-reach vulnerabilities. To deep-dive into target binary, some software security studies have provided patching-based fuzzing approaches. For example, checksum-aware directed fuzzing tool TaintScope can detect nested complex checks based on fine-grained dynamic taint tracking and branch profiling [35]. Furthermore, transformational fuzzing T-Fuzz [23] has been proposed recently. It detects complex checks and bypasses those checks by easily flipping the condition of the jump instruction [23]. However, the crash analyzer of the T-Fuzz cannot work well for all binaries, and it develops a transformation explosion problem if the true bug is hidden in depths of the target program [6,23].

The state-of-the-art hybrid fuzzing. A decade ago, the hybrid fuzzing approach was proposed to achieve high code coverage and detect more vulnerabilities. Recently, this approach demonstrated its highly efficient performance in software bug detection. For instance, Driller [18] won the DARPA Cyber Grand Challenge by its hybrid bug excavation approach that used AFL and selective concolic execution. Another state-of-the-art hybrid fuzzer is QSYM [22], which implemented fast concolic execution with AFL, achieved high efficiency performance by integrating symbolic emulation with native execution. Yet another type of hybrid fuzzing technique is DeepFuzz [47], which combines dynamic symbolic execution with AFL to trigger bugs in deep layers of the binary. In contrast, to achieve a high code coverage and reach to the vulnerabilities that are hidden in-depth under the nested complex roadblock checks, "*transformational hybrid fuzzer*" DeepDiver changes the hybrid fuzzing philosophy. It removes the nested complex roadblocks to executing new paths and deep dives into the binary. As a result, it will be able to achieve outstanding results in bug coverage.

7. Conclusions

It is rather difficult to bypass nested complex checks in programs for mutational or symbolic execution-based fuzzing techniques and thus, hard-to-reach bugs will be left uncovered.

In this paper, we introduced a new transformational hybrid fuzzing approach. We implement a scalable, lightweight hybrid fuzzing technique named DeepDiver, which can detect roadblock checks and negate them to execute new paths. According to the evaluation results, our implementation is scalable to test real-world programs and can deep-dive into complex binaries. Moreover, DeepDiver performed much better than the most modern state-of-the-art hybrid fuzzers, i.e., QSYM and DeepFuzz, as well as other types of fuzzers in LAVA-M dataset and real-world applications. On average, DeepDiver trigger bugs 32.2% and 41.6% faster than QSYM and AFLFast, respectively. Ultimately, we believe that our hybrid fuzzing design will contribute to the advancement of software vulnerability testing processes.

Author Contributions: F.R. contributed the ideas, conducted the experiments and wrote the paper; J.K. provided resources and database of vulnerable programs; J.Y. supervised the whole paper including paper organization and proofread. 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-2020-2018-0-01423) supervised by the IITP (Institute for Information & Communications Technology Planning & Evaluation). Also, 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).

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

References

1. Chen, P.; Hao, C. Angora: Efficient fuzzing by principled search. In Proceedings of the 39th IEEE Symposium on Security and Privacy, IEEE, San Francisco, CA, USA, 21–23 May 2018.

2. Mu, D.; Cuevas, A.; Yang, L.; Hu, H.; Xing, X.; Mao, B.; Wang, G. Understanding the reproducibility of crowd-reported security vulnerabilities. In Proceedings of the 27th USENIX Conference on Security Symposium, USENIX Association, Baltimore, MD, USA, 15–17 August 2018; pp. 919–936.
3. Zalewski, M. American Fuzzy Lop (AFL), README. Available online: http://icamtuf.coredump.cx/afl/ (accessed on 29 October 2019).
4. Hocevar, S. zzuf:multi-purpose Fuzzer. Available online: http://caca.zoy.org/wiki/zzuf (accessed on 9 November 2019).
5. Oehlert, P. Violating assumptions with fuzzing. *IEEE Secur. Priv.* **2005**, *3*, 58–62.
6. T-Fuzz Source Code. Available online: https://github.com/HexHive/T-Fuzz (accessed on 9 November 2019).
7. Month of Kernel Bugs. Available online: https://jon.oberheide.org/mokb/ (accessed on 10 December 2019).
8. Month of Browser Bugs. Available online: http://browserfun.blogspot.com (accessed on 10 December 2019).
9. The Heartbleed Bug. Available online: http://heartbleed.com/ (accessed on 10 December 2019).
10. Serebryany, K. OSS-Fuzz—Google's Continuous Fuzzing Service for Open-Source Software. Available online: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/serebryany (accessed on 10 December 2019).
11. Takanen, A.; DeMott, J.; Miller, C.; Kettunen, A. In *Fuzzing for Security Testing and Quality Assurance*, 2nd ed.; Artech House: London, UK, 2018; pp. 1–2; ISBN 9781608078509.
12. Sutton, M.; Greene, A.; Amini, P. Fuzzing limitations and expectations. In *Fuzzing: Brute Force Vulnerability Discovery*; Addison-Wesley Professional: Crawfordsville, IN, USA, 2007; pp. 29–32; ISBN 0321446119.
13. Serebryany, K. LibFuzzer–a Library for Coverage-Guided Fuzz Testing. Available online: http://llvm.org/docs/LibFuzzer.html (accessed on 9 November 2019).
14. Fuzzing Tutorial: What Is, Types, Tools and Example. Available online: https://www.guru99.com/fuzz-testing.html (accessed on 13 November 2019).
15. Majumdar R.; Sen, K. Hybrid Concolic Testing. In Proceedings of the 29th International Conference on Software Engineering (ICSE'07), Minneapolis, MN, USA, 20–26 May 2007; pp. 416–426.
16. Li, Y.; Chen, B.; Chandramohan, M.; Lin, S.W.; Liu, Y.; Tiu, A. Steelix: Program-State Based Binary Fuzzing. In Proceedings of the 11th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE), Paderborn, Germany, 4–8 September 2017.
17. Pak, B.S. Hybrid Fuzz Testing: Discovering Software Bugs via Fuzzing and Symbolic Execution. Master's Thesis, School of Computer Science Carnegie Mellon University, Pittsburgh, PA, USA, 2012.
18. 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 2016 Network and Distributed System Security Symposium, San Diego, CA, USA, 21–24 February 2016.
19. Sen, K.; Marinov, D.; Agha, G. CUTE: A Concolic Unit Testing Engine for C. In Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE-13), Lisbon, Portugal, 5–9 September 2005.
20. Cadar, C.; Sen, K. Symbolic execution for software testing: Three decades later. *Commun. ACM* **2013**, *56*, 82–90.
21. Wang, X.; Sun, J.; Chen, Z.; Zhang, P.; Wang, J.; Lin, Y. Towards optimal concolic testing. In Proceedings of the 40th International Conference on Software Engineering, Gothenburg, Sweden, 27 May–3 June 2018.
22. 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
23. 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.
24. Ognawala, S.; Hutzelmann, T.; Psallida, E.; Pretschner, A. Improving function coverage with munch: A hybrid fuzzing and directed symbolic execution approach. Proceedings of the 33rd Annual ACM Symposium on Applied Computing, Pau France, 9–13 April 2018.
25. Cybersecurity 101: What You Need to Know about False Positives and False Negatives. Available online: https://www.infocyte.com/blog/2019/02/16/cybersecurity-101-what-you-need-to-know-about-false-positives-and-false-negatives/ (accessed on 3 November 2019).

26. 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.
27. 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.
28. Swiecki, R. Honggfuzz. Available online: http://code.google.com/p/honggfuzz (accessed on 10 December 2019).
29. Serebryany, K. Continuous fuzzing with libfuzzer and addresssanitizer. In Proceedings of the 2016 IEEE Cybersecurity Development (SecDev), Boston, MA, USA, 3–4 November 2016; pp. 157–157.
30. Checksum: From Wikipedia Encyclopedia. Available online: http://en.wikipedia.org/wiki/Checksum. (accessed on 6 November 2019).
31. Checksum. Available online: https://searchsecurity.techtarget.com/definition/checksumyou (accessed on 3 November 2019).
32. 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 Febuary–1 March 2017.
33. Circumventing Fuzzing Roadblocks with Compiler Trans-Formations. Available online: https://lafintel.wordpress.com/ (accessed on 11 December 2019).
34. Hauster, M.; Heiko E.; Fioraldi, A. American Fuzzy Lop Plus Plus (AFL++). Available online: https://github.com/vanhauser-thc/AFLplusplus (accessed on 11 December 2019).
35. Wang, T; Wei, T; Gu, G; Zou, W. TaintScope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection. In Proceedings of the 2010 IEEE Symposium on Security and Privacy, Berkeley/Oakland, CA, USA, 16–19 May 2010.
36. Radare2-Reverse-Engineering Framework, Available online: https://rada.re/r/docs.html (accessed on 3 October 2019).
37. r2pipe:radare2 Python Scripting, Available online: https://r2wiki.readthedocs.io/en/latest/home/radare2-python-scripting/ (accessed on 3 October 2019).
38. Introduction of r2pipe, Available online: https://www.peerlyst.com/posts/r2con-2018-j-geno (accessed on 3 October 2019).
39. Böhme, M.; Pham, V.T.; Roychoudhury, A. Available online: https://github.com/mboehme/aflfast (accessed on 3 November 2019).
40. Lyu, C.; Ji, S.; Zhang, C. MOPT: Optimized Mutation Scheduling for Fuzzers. Available online: https://github.com/puppet-meteor/MOpt-AFL (accessed on 10 December 2019).
41. Hsu, C.; Wu, C. InsTrim: Lightweight Instrumentation for Coverage-Guided Fuzzing. Available online: https://github.com/csienslab/instrim (accessed on 10 December 2019).
42. Helin, A. A General-Purpose Fuzzer. Available online: https://gitlab.com/akihe/radamsa (accessed on 3 November 2019).
43. 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.
44. 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 Febuary 2008.
45. Yun, I. QSYM: A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing. Available online: https://github.com/sslab-gatech/qsym (accessed on 1 September 2019).
46. LAVA: Large-Scale Automated Vulnerability Addition. Available online: https://github.com/panda-re/lava (accessed on 5 September 2019).
47. Bottinger, K.; Eckert, C. Deepfuzz: Triggering vulnerabilities deeply hidden in binaries. 13th Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA), San Sebastián, Spain, 7–8 July 2016.
48. Tracer: Utilities for Generating Dynamic Traces. Available online: https://github.com/angr/tracer (accessed on 5 October 2019).

49. Shoshitaishvili, Y.; Wang, R.; Salls, C.; Stephens, N.; Polino, M.; Dutcher, A.; Grosen, J.; Feng, S.; Hauser, C.; Kruegel, C.; Vigna, G. Sok: (state of) the art of war: Offensive techniques in binary analysis. In Proceedings of the 2016 IEEE Symposium on Security and Privacy (SP), San Jose, CA, USA, 22–26 May 2016; pp. 138–157.
50. Gan, S.; Zhang, C.; Qin, X.; Tu, X.; Li, K.; Pei, Z.; Chen, Z. CollAFL: Path sensitive fuzzing. In Proceedings of the IEEE Symposium on Security and Privacy. San Francisco, CA, USA , 20–24 May 2018; pp. 660–677.
51. Binutils Source Code. Available online: https://ftp.gnu.org/gnu/binutils/ (accessed on 23 November 2019).
52. libksba Source Code. Available online: https://fossies.org/linux/privat/libksba-1.3.5.tar.gz/ (accessed on 23 November 2019).
53. libjpeg Source Code. Available online: https://www.ijg.org/files (accessed on 23 November 2019).
54. libarchive Source Code. Available online: https://libarchive.org/downloads/ (accessed on 23 November 2019).
55. tcpdump Source Code. Available online: http://www.tcpdump.org/release/ (accessed on 23 November 2019).
56. poppler Source Code. Available online: https://poppler.freedesktop.org/releases.html (accessed on 23 November 2019).