# HDBFuzzer–Target-oriented Hybrid Directed Binary Fuzzer

Yingchao Yu
State Key Laboratory of Mathematical Engineering and Advanced Computing, Wuxi, China
yuych830305@163.com

Xiaojun Qin
State Key Laboratory of Mathematical Engineering and Advanced Computing, Wuxi, China
qxj@163.com

Shuitao Gan
State Key Laboratory of Mathematical Engineering and Advanced Computing, Wuxi, China
ganshuitao@gmail.com

## ABSTRACT

In this paper, we propose a target-oriented hybrid directed binary fuzzer (HDBFuzzer) to solve the vulnerability confirmation problem based on binary code similarity comparison. HDBFuzzer combines macro function level direction fuzzing and micro path-constraint directed solving. For some branches with simple or loose constraints, it still uses directed mutation of the directed fuzzing to penetrate while for some really hard-to-penetrate constraints, it resorts to guided concolic execution. At the same time, in order to improve the efficiency of constraint solving, we propose a constraint solving method based on "path abstraction", which approximates the solution space by the linear expression and generates effective input utilizing the highly-effective sampling method towards the linear space. Then, under the guidance by the directed greybox fuzzing, HDBFuzzer can generate input that can quickly reach the vulnerable code region and finally crash the program under the test to confirm the vulnerability hidden in the binary program. We evaluate HDBFuzzer against AFLGo-B and QSYM on LAVA-M dataset and ten real-world programs, and the results show that HDBFuzzer is superior to AFLGo-B and QSYM on the bug discovery, bug reproduction and target reaching capabilities.

## CCS CONCEPTS

• **Security and Privacy**; • **Systems security**; • **Vulnerability management**; • **Penetration testing**;

## KEYWORDS

Binary program, Target site, Directed Greybox Fuzzing, Symbolic execution, Vulnerability confirmation

**ACM Reference Format:**
Yingchao Yu, Xiaojun Qin, and Shuitao Gan. 2021. HDBFuzzer–Target-oriented Hybrid Directed Binary Fuzzer. In *The 5th International Conference on Computer Science and Application Engineering (CSAE 2021), October 19–21, 2021, Sanya, China.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3487075.3487124

## 1 INTRODUCTION

In recent years, binary code similarity matching technology and its application on the field of vulnerability search have been gradually got great attention and made great progress in the industry and academia, especially in the field of the embedded device firmware vulnerability search. The basic idea is that given a known vulnerability function such as a CVE function and a binary program under the test (PUT), identify in the PUT the most similar function to the vulnerability function as the vulnerable function and verify whether the identified vulnerable function indeed contains a real vulnerability. The former can be accomplished by binary code similarity matching tools [1-4]. For the latter, however, the existing methods are almost based on manual validation, which are time-consuming and error-prone. Especially in large-scale binary firmware vulnerability search, it's almost not realistic to confirm whether the vulnerable function indeed contains real vulnerability only just by means of manual analysis one by one.

Fuzzing test [5] is an effective way to find vulnerabilities in a program. Traditional Coverage-guided Greybox Fuzzing (CGF) tools aim to cover as much program state as possible in the given time budget to find the path that might crash the program. However, higher coverage doesn't necessarily mean more vulnerabilities can be found, as fuzzers will blindly probe all possible program states rather than focus on the functions that are more likely vulnerable. Therefore, they are not very effective when applied to vulnerability confirmation based on binary code similarity detection. Recently, Directed Greybox Fuzzing (DGF) has been proposed, such as AFLGo [6] and Hawkeye [7], which focus on driving the testing towards specific program locations (which are called target sites) and focus fuzzing on such target sites. Compared with the traditional CGF, DGF is more suitable for the application scenario of firmware vulnerability search with binary code similarity matching, if the location information of the target sites can be obtained. However, DGF inherits the same inherent defects of the traditional fuzzing as CGF, that's to say, it's difficult to generate test inputs that can go through the path protected by complex constraints to probe into vulnerable code regions which can trigger the PUT crashes. Recent efforts on hybrid fuzzing, such as Driller [8] or QSYM [9], combine fuzzing and symbolic execution to deeply probe into the program, taking full advantage of the fuzzing's high-speed testing capabilities and the symbolic execution's effective constraint solving capabilities to generate test inputs that can passthrough paths protected by complex constraints to trigger the potential vulnerabilities in the PUT. However, they still focus on achieving higher code coverage, without the directional guidance of the target sites, and therefore can't be directly applied to the binary code similarity-based vulnerability confirmation problem.

In this paper, a Hybrid Directed Binary Fuzzer (HDBFuzzer) driven by target site is proposed to solve the vulnerability confirmation problem based on the binary code similarity matching. Its core idea is to use DGF to carry out high-speed fuzzing test towards


*CSAE 2021, October 19–21, 2021, Sanya, China*

ACM ISBN 978-1-4503-8985-3/21/10…$15.00
https://doi.org/10.1145/3487075.3487124

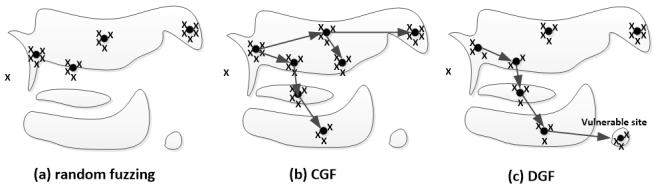


**Figure 1: Random Fuzzing vs. CGF vs. DGF.**

the target sites, and at the same time, concolic execution is used to help solving some complex constraints alongside the path to the target sites so as to generate test inputs that can pass through some specific branches for DGF to use. Compared to the conventional hybrid fuzzer, HDBFuzzer is a binary-oriented, target sites-driven, hybrid directed fuzzer, aiming to generate inputs that can reach the target sites as quickly as possible so as to focus resources on the vulnerable code regions for concentrated testing and confirm whether there are real vulnerabilities in the vulnerable code regions.

## 2 BACKGROUND AND MOTIVATION

Automated vulnerability detection can be viewed as the process of searching the input space of the PUT to identify which input can trigger a bug. Since the input space of real-world programs is extremely very large, the goal of the automated vulnerability detection technology is to search for interesting inputs that can trigger new program state (new code line or new path) automatically and intelligently.

### 2.1 Random Fuzzing vs. CGF vs. DGF

As shown in Figure 1, we divide the automated vulnerability detection technologies into three categories according to the different ways of detecting the input space: random fuzzing, CGF and DGF. Random fuzzing will blindly probe the input space, as shown in Figure 1(a), they mainly generate new inputs around the initial seeds, so the new generated interesting inputs (thus the new program states detected) are very limited. CGFs aim to cover as many paths as possible in order to discover as many crashes as possible, if they exist. However, they are still "blind", although they provide feedback as dynamic guidance to determine which inputs are interesting inputs and should be included in the extended corpus. As special type of CGF, DGFs will specify one or more target sites (that is, the preferred program location(s) which DGFs will drive the seeds to, and usually set to the crashing point(s) and its/their associated location(s)), and then drive the fuzzer towards the target site(s) as far as possible for concentrated testing. Compared to CGFs, DGFs will spent most time on detecting the code paths towards the target site(s) and won't waste any more resources on the unrelated paths. Therefore, DGFs are "directed" and "targeted".

### 2.2 Hybrid Fuzzing

Although many bugs have been discovered by CGF over the years, however, there are still situations that the mutation strategy in CGF does not work. For example, CGF will get stuck on checking on some magic bytes, and it has been proved that it's almost impossible to generate inputs that can passthrough the magic bytes just by random mutation. Magic bytes checking can occur in the very early stage of the program execution, fuzzers can quickly get stuck and stop generating the interesting inputs to deeply probe the program path. While systematic methods, such as symbolic or concolic execution, can systematically detect input space targeting specific program path and generate inputs that cover these paths with the help of the Satisfiability Modulo Theories (SMT) solver [10], despite the performance overhead is very high. As a result, it's widely used by the researchers as an aid to the fuzzing, hence introduces the hybrid fuzzing. Unlike traditional symbolic execution (SE)/ concolic execution (CE), the SE/CE in hybrid fuzzing will not probe for branch path, it just solves the constraints, sends the generated input to the fuzzer, and then, exit. Then the fuzzer will continue to probe the new input space to discover the bugs hidden in the deeper codes. In other words, hybrid fuzzing mainly relies on the fuzzer to probe the program, reducing the frequency of calling SE/CE, thus alleviating the possible state explosions.

### 2.3 Analysis of existing tools and our motivation

**Analysis of existing tools**. We list some typical DGF and hybrid fuzzing works in recent years and their technical characteristics in Table 1. AFLGo [6] first proposed a CFG-based distance to evaluate the proximity between seed execution and multiple target sites, as well as a simulated anneal-based energy scheduling. Hawkeye [7] improved the accuracy by introducing a seed selection heuristic algorithm based on target coverage and self-adaptive mutation. ParmeSan [11] proposed an accurate dynamic CFG building method and a two-staged directed fuzzing strategy to effectively guide the fuzzing to all the targets of interest. FuzzGuard [12] was a deep learning-based work, which predicted the reachability of the inputs (i.e., whether the target can be reached) before executing the target program and then filtered out the unreachable ones to improve the performance of the fuzzer (that is AFLGo in FuzzGuard). Driller [8] was the first hybrid fuzzing framework. It used AFL as CGF and angr for SE interchangeably during the testing. When AFL can't generate new interesting inputs in a given time budget, it will automatically switch to the angr for constraint solving. It can successfully and easily solve obstacles like magic bytes comparison. Once the condition check was passed, Driller will restart the fuzzer. QSYM [9] addressed the performance bottlenecks of existing CEs by tightly integrating dynamic binary translation and native symbolic execution, making it extensible to discover bugs in real-world programs.

It can be seen from Table 1 that most of the existing DGFs are source code-oriented, which need to analyze the source code to extract the target site(s) and calculate the distance between each seed and the target site(s) to guide the directed fuzzing in the second stage. Hybrid fuzzing can be binary oriented, but most of them are still aiming to improve code coverage, not considering the directness of the testing. On the other hand, during the actual testing, it was found that a large number of inputs could not reach the potential buggy code. Even with the most state-of-the-art DGF such as AFLGo, on average more than 91.7% of the inputs can't reach the buggy code [12]. This large number of unreachable inputs will

**Table 1: Summary of Existing Typical DGF and Hybrid Fuzzing Tools**

| Category | Tools | Literature | Technologies | binary | Open-source | problem solved |
|---|---|---|---|---|---|---|
| DGF | AFLGo | CCS17 | AFL+ simulated anneal | no | yes | directness |
| | Hawkeye | CCS18 | AFL-like+ simulated anneal | no | no | directness |
| | ParmeSan | usesec20 | libfuzzer+ | no | yes | directness |
| | FuzzGuard | usesec20 | AFLGo+ unreachable inputs filtered out | no | no | directness |
| hybrid Fuzzing | Driller | NDSS16 | AFL↔selective symbolic execution | yes | yes | coverage |
| | QSYM | usesec18 | hybrid fuzzing & dynamic SE | yes | yes | coverage |
| | DrillerGo | CCS19-poster | hybrid directed fuzzing | yes | no | directness |

waste a log of fuzzing time, especially when they are applied to hybrid fuzzing. If symbolic execution passes lots of crash-independent inputs to the fuzzer, the fuzzer will delve into many meaningless paths, which may be resulting in a vicious cycle causing the fuzzing queue growing and further greatly wasting system resources and time budget. The reason behind is that the random mutation in existing DGF tools makes it difficult to handle complex constraints. Even in hybrid fuzzing, it's difficult or even impossible to solve the input that satisfies the very complex constraints alongside the program path from the entry point to the buggy code in a given time budget.

**Our Method**. In order to overcome the challenges faced by the existing DGF or hybrid fuzzing tools in addressing problem of the vulnerability confirmation based on binary code similarity matching, a hybrid directed binary fuzzer HDBFuzzer is proposed in this paper. Compared with regular hybrid fuzzer, HDBFuzzer is target-oriented, directed, hybrid binary fuzzer. The goal of HDBFuzzer is to generate inputs that can reach the target site(s) as quickly as possible so as to trigger hard-to-reach bugs in the binary and further verify that the vulnerable code region indeed contains a bug. Due to the lack of low-level penetration testing on local path conditions, function-level directed fuzzing may get stuck on some code and fail to reach the target site(s), so, HDBFuzzer combines micro path constraint solving and macro function-level directed fuzzing together to quickly and efficiently generate inputs that can reach the target site(s). Particularly, instead of blindly solving all branch constraints, HDBFuzzer will only select those branches that are really difficult for the fuzzer to break through and submit them to the concolic execution for constraint solving. Specifically, branches with simple or loose constraints will be still explored by fuzzing through directed mutation, while branches that are really difficult to break through for the fuzzer will be put forward to the concolic execution. At the same time, to improve the efficiency of constraint solving, we propose one constraint solving method based on "path abstraction". This method will approximate the solution space of the constraints by linear expressions and generate valid inputs by means of efficient sampling towards the approximated linear space. Then it will generate inputs that can reach vulnerable code regions as quickly as possible under the directed fuzzing, and eventually generate crashing-inputs, confirming the vulnerability in the binary, if any.

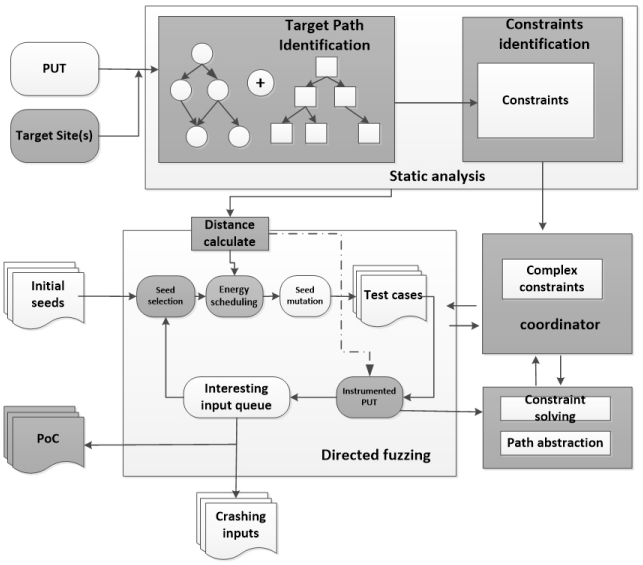


**Figure 2: HDBFuzzer Overview.**

## 3 HYBRID DIRECTED BINARY FUZZER: HDBFUZZER

The overview of HDBFuzzer is shown in Figure 2. It consists of five components: target site(s) identification, static analysis, directed fuzzing, hard constraint solving based on path abstraction and coordinator. Here, the target site(s) identification is used to identify the vulnerable code regions (target site(s)) in the target binary. The static analysis module is used to identify the set of the target path and the set of the constraints on the target path(s). Directed fuzzing is used to perform directed greybox fuzzing. The constraint solving component based on the path abstraction is used to get the solution of the hard constraints which are indeed hard to break through. The coordinator is used to coordinate the interaction between the directed fuzzing and constraint solving.

### 3.1 Target Site(s) Identification

Given a PUT and a related vulnerability (CVE or patch), the goal of target site(s) identification is to identify the potential vulnerable code region, also known as target site(s) in this paper. As shown in Figure 3, given a CVE and a PUT, we can output the potential vulnerable target functions through a well-trained neural

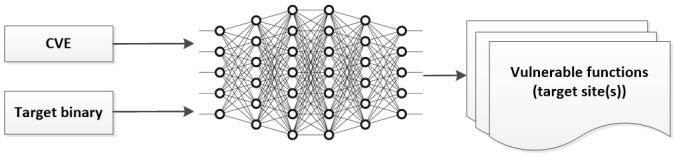


**Figure 3: Target Site(s) Identification Process.**

network sorted by their similarity scores with the CVE, denoted as {(func1,score1,address1), (func2, score2, address2), . . ., (funcn, scoren, addressn)}, where funci, scorei and addressi denote the vulnerable function, the similarity score between the vulnerable function and the CVE function and the relative address of the vulnerable function in the target binary, respectively. Since we focus on how to generate the input for reaching the target site(s) as quickly as possible to confirm there is indeed a true vulnerability around the target site(s), so we will not illustrate the technical details of the process of the target site(s) identification too detailly.

Target Path and Constraint Identification.

The process of the target path and constraint identification is shown in Algorithm 1. It first builds the call graph (CG) of P according to the binary program P and its entry point Entrypoint (line 2). Then, for each function in CG, it builds its control flow graph (CGF, line 4) and inter-process control flow graph of P combing CG and CFGs (line 5). For each target target in target site(s) T, backward pathfinding will be used on P's ICFG to find the path from Entrypoint to the target. In the following process of the constraint solving based on path abstraction, only when the address of the hard branched constraint is included in the collected addresses by the backward pathfinding module gathers here, HDBFuzzer will solve it.

**Algorithm 1** Target path identification algorithm

Input:*P*: binary program, *Entrypoint*: entry point, *T*: target sites
Output: TP: target path
function ExtractTargetPaths(*P*):

```
1.  TP ← { }
2.  CG ← BuildCallGraph(P, Entrypoint)
3.  for each function in CG:
4.      CFG ← BuildControlGraph(function, function_entry)
5.      ICFG ← CG + CFGs
6.  for each target ∈ T :
7.      path ← BackSlicking(target, Entrypoint, ICFG)
8.      c_p ← constraints_on_path(path)
9.     TP ← TP ∪{path,  c_p}
10.  return TP
```

## 3.2 Complex Constraint Solving Based on Path Abstraction

When the backward pathfinding module completes the target path identification, the dynamic analysis module starts to run. The dynamic analysis module combines the function-directed fuzzing and path constraint guided concolic execution in a complementary and cooperative way. On the one hand, since the directed fuzzing may be stuck in some complex code and fail to reach the target function, path constraint solving is introduced to assist the fuzzer to get through these difficult constraints. On the other hand, different from blindly solving all the path condition branches, we only select branches that are really difficult for the fuzzer to break through to the concolic execution so as to alleviate the possible path explosion and low performance in solving constraints.

**Difficult constraints identification**. When the fuzzer gets stuck, that is, when the fuzzer stops moving forward in a certain period of time, Dirller [8] will start concolic execution. In its implementation algorithm, Driller runs AFL as its fuzzing component, when the pending_favs in AFL drops to 0, Driller considers that it meets a difficult constraint, then calls concolic execution to solve it. However, just as DigFuzz [13] mentioned, this "conservative" strategy is problematic because it's not a good identifier for starting the concolic execution as fuzzer gets stuck for a period of time, which is more obvious for the real-world programs, even after testing a few hours later, pending_favs may not drop to 0. In this case, it will not call the concolic execution during the fuzzing, that is, the hybrid fuzzer is degraded to a pure fuzzer. Unlike Driller, QSYM [9] will synchronize the inputs from the fuzzer with its concolic execution. However, during the process of testing towards real-world programs by QYSM, we found that only 25.2% of the inputs generated by QSYM will be considered interesting and adopted by AFL; the other 74.8% will be considered uninteresting and discarded. The reason is that fuzzers like AFL can quickly cover many branches with loose conditions such as if(x==0x50) or if(x≥56), in which cases, concolic execution will waste a lot of time on solving these useless tasks.

In this paper, we select a relative neutral strategy, choosing the most difficult constraints for he fuzzer to break to concolic execution to generate inputs that can pass branches protected by complex and hard constraints, such as magic bytes branches like if(x=0xdeadbeef) or nested condition branches like if(x+y>10) { if (x>5). . ..}, so as to reach the target site(s) as quickly as possible. As stated earlier, only when the address of a branch unit is included in the addresses collected by the backward pathfinding module and the fuzzer is unable to break through this branch unit for a period of time, the directed concolic execution module will be called for constraint solving.

Constraint solving based on path abstraction. In traditional hybrid fuzzing, concolic execution will directly call the constraint solver to get a feasible solution, which is limited by the performance bottleneck of constraint solving, making it difficult to achieve efficient path detection. Unlike this, we select to construct path abstraction on these unsolved branches to approximately describe the search space of the feasible solutions of all the path constraints so as to accelerate the solving of the subsequent path constraints. Specifically, before solving the target constraints, we will first construct the simplified form of the target path constraint by the path abstraction of its precursors. It's simpler and easier to solve for the simplified target path constraint than the original one. If the simplified path constraint can not be satisfiable, the target path constraint must not be satisfiable, and the current path is unreachable. If the simplified path constraint can be satisfied, then it can reduce the solution space of the target path constraint.

We will use $pc(p)$ and $\widehat{pc}(p)$ to indicate the path constraint and its path abstraction on path $p$, and $p$=$p_1$+$p_2$, where $p_1$, $p_2$ indicates the prefix and the remaining path, respectively. Before solving $pc(p)$, we will solve the simplified constraint abstracted by the existing prefix path, that is, $\widehat{pc}(p_1) \wedge pc(p_2)$. In general, it's less complex for the simplified form than the original form, so it's easier to solve. If $\widehat{pc}(p_1) \wedge pc(p_2)$ is unsatisfiable, the path can be pruned immediately, because $pc(p_1) \wedge pc(p_2)$ must be unsatisfiable. Else, if $\widehat{pc}(p_1) \wedge pc(p_2)$ is satisfiable, we will continue to solve $\widehat{pc}(p_1) \wedge pc(p_1) \wedge pc(p_2)$, that is, $\widehat{pc}(p_1) \wedge pc(p)$. Although it seems more complex than $pc(p)$, it's actually much easier to solve because the path abstraction $\widehat{pc}(p_1)$ reduces the solution space of the path constraint. Since the path abstraction contains the calculated linear expressions that appear in the path constraint, the connection between the path constraint and the path abstraction directly reduces the input's search space to solve.

## 3.3 Directed Fuzzing

*3.3.1 Distance Calculation.* Distance calculation is the core of the DGFs. We will list the function distance, basic block distance and the seed distance calculation in the following.

**Function distance**. The function distance between $n$ and $n'$ is defined as the shortest distance between the two functions on the CG. As in formula 1, the function distance of the function $n$ ($i.e.$ $D_f$) is defined the harmonic mean of the function distance between the function $n$ and the reachable target function $T_f$.

$$D_f\left(n, T_f\right) = \begin{cases} undefined, & if\ no\ path\ from\ f\ to\ T_f \\ d, & otherwie \end{cases} \tag{1}$$

where, $d$ is the harmonic mean between $f$ and all the reachable objective functions $T_f$.

**Basic block distance**. The target distance at the basic block level, $D_{bb}(m, T_b)$, is defined as the distance from each block $m$ to the target block $T_b$. It can be calculated across different functions, which are defined as the harmonic average of each function that calls the block.

$$D_{bb}\left(m, T_b\right) = \begin{cases} 0 & if\ m \in T_b \\ c \cdot \min\limits_{n \in (m)} \left(D_f\left(n, T_f\right)\right) & if\ m \in T \\ \left[\sum_{t \in T} \left(D_{bb}\left(m, t\right) + D_{dd}\left(m, T_b\right)\right)^{-1}\right]^{-1} & otherwise \end{cases} \tag{2}$$

where, $N(m)$ is the set of the function that calls the basic block m, T is the set of basic blocks included in the function that calls m, and c is set to 10.

**Seed distance.** The seed distance is defined as the average of the distances between each executed basic block and the target:

$$dist_{seed} = \frac{\sum_{m \in N(seed)} D_{bb}\left(m, T_b\right)}{|N\left(seed\right)|} \tag{3}$$

where, $N(seed)$ is the set of the basic blocks the seed *seed* executes and it has one path to the target block $T_b$.

*3.3.2 Directed Fuzzing Algorithm.* Algorithm 2 describes the process of the directed fuzzing and the differences from the AFL are underlined. To start the fuzzing, directed fuzzing firstly applies each initial input I on P, recodes the corresponding execution trace and calculates the distance between each input and the target according to the execution trace (line 2). The same procedure is performed for each variant input i' (line 7). If the execution reaches T, that is, the distance between i' and T is 0, then i' will be added to H (line 8 and 9). At the beginning of the fuzzing, the directed fuzzing will sort the inputs in the queue according to the distance between each input and the target in order to ensure the input closer to the target can be mutated earlier (line 4). If the execution trace of the variation input covers new branches or hits one branch more than one time, the variation will be added to the queue I. To Ensure the input with shorter distance to the target branch can still be mutated earlier, the directed fuzzing will insert each newly generated and interesting variation input to the position after i in the queue I in ascending order of their distance in order to make i' can be selected quickly and can be mutated in the subsequent internal loop iteration. Each time the queue traverses the tail, it will sort the queue again (line 11). Because the insertion operation only guarantees that the newly generated input will be inserted after its parent input by distance, so the entire queue may be unordered after the insertion.

**Algorithm 2** Improved Directed Fuzzing Greybox Fuzzing Algorithm

Input:P: binary program, I: initial seeds, T: target site(s)
Output:H: set of inputs that can reach T in P; B: crashing-inputs set
$\boldsymbol{H} = \emptyset$;

1. **for** $\boldsymbol{i}$ **in** $\boldsymbol{I}$ **do**
2. $\boldsymbol{d} \leftarrow \boldsymbol{run\_target}(\boldsymbol{P}, \boldsymbol{T}, \boldsymbol{i})$
3. **while** ***timeout not excedded*** **do**
4. $\boldsymbol{I'} \leftarrow \boldsymbol{sort\_by\_distance}(\boldsymbol{I}, \boldsymbol{d})$
5. **for** $\boldsymbol{i}$ **in** $\boldsymbol{I'}$ **do**
6. $\boldsymbol{i'} = \boldsymbol{mutate\_input}(\boldsymbol{i})$
7. $\boldsymbol{d'} \leftarrow \boldsymbol{run\_target}(\boldsymbol{P}, \boldsymbol{T}, \boldsymbol{i'})$
8. **if** $\boldsymbol{d'} == 0$ **then**
9. $\boldsymbol{H} = \boldsymbol{H} \cup \{\boldsymbol{i'}\}$
10. **if** $\boldsymbol{is\_interesting}(\boldsymbol{i'})$ **then**
11. $\boldsymbol{I} = \boldsymbol{I} \cup \{\boldsymbol{i'}\}$
12. **if** $\boldsymbol{trigger\_crash}(\boldsymbol{i'})$ **then**
13. $\boldsymbol{B} = \boldsymbol{B} \cup \{\boldsymbol{i'}\}$
14. $\boldsymbol{i} = \boldsymbol{next}(\boldsymbol{I'}, \boldsymbol{i})$
15. end for
16. **end**
17. return H, B

# 4 IMPLEMENTATION AND EVALUATION

## 4.1 Implementation

HDBFuzzer is implemented based on the AFL(2.52b), QEMU(2.10.0), IDA Pro(v7.5) and QSYM. IDA Pro is used to extract the ICFG and calculate the distance. AFL-QEMU is modified to trace the covered targets and calculate the seed score and the energy scheduling function dynamically. When AFL-QEMU encounters a branch that is difficult to cover, HDBFuzzer will call the Z3's SMT-opt algorithm to calculate the path abstraction and solve the branch. Information such as target location, time budget and fuzzing status are shared between AFL and QSYM. All the tools in this paper will use the same empty file or the valid file provided by the developer as the initial seed.

**Table 2: LAVA-M Bugs Found by Different Fuzzers**

| Tools | base64(44 bugs) | md5sum(57 bugs) | uniq(28 bugs) | who(2136 bugs) | total(2265) |
|---|---|---|---|---|---|
| AFLGo-B | 16(36.36%) | 35(61.40%) | 15(53.57%) | 1058(49.53%) | 1124 |
| QSYM | 44(100%) | 57(100%) | 28(100%) | 1353(64.42%) | 1482 |
| HDBFuzzer | 44(100%) | 57(100%) | 28(100%) | 2018(94.48%) | 2147 |

**Settings.** Following the evaluation criteria in the literature [14], the same fuzzing configuration and hardware resources configuration are used for all the experiments in this paper. All experiments are conducted on an assembled Linux desktop equipped with Intel(R) Core(TM) i7-8700K CPU @ 3.70GH, 12 virtual cores, 16GB, Ubuntu 16.04, with python 3.7.

**Baseline fuzzers**. As shown in Table 1, only AFLGo [6], Driller [8], QSYM [9] and ParmeSan [11] in existing DGF and hybrid fuzzing tools are open-sourced. However, AFLGo and ParmeSan are both source code-oriented, and ParmeSan relies on source code-level memory address sanitizer to extract target sites, which is difficult to change it to support binary programs due to the lack of effective binary memory address sanitizer at present. Therefore, we select AFLGo-B (we rewrite the AFLGo to support binary program) and QSYM (which has been shown to be superior to Driller) as the baseline fuzzers to compare with HDBFuzzer.

**Dataset and Evaluation Criteria.** To evaluate the effectiveness of the fuzzers, we conduct experiments on the widely used LAVA-M dataset [15] and some real-world programs to evaluate the vulnerability discovery, bug reproduction and target reachability capabilities. In this paper, we use TTE (time-to-exposure, the first time finding the input that triggers the bug) and the success runs when the input triggers the bug to evaluate the bug reproduction capability of the fuzzers. And we use TTR (time-to-reach, the first time reaching the target site) to evaluate the target reachability of the fuzzers. The smaller the TTE and the more success runs, the better the fuzzer. The smaller the TTR, the better the fuzzer.

## 4.2 Evaluation

*4.2.1 Bug Discovery and Bug Reproduction on LAVA-M Dataset.* LAVA-M dataset is an artificial evaluation dataset for bug detection. There are many complex bugs in this dataset and most fuzzing tools use this dataset to evaluate the performance of their proposed methods. In the LAVA-M dataset, each bug corresponds to a unique identifier. When the bug is triggered by a fuzzer, it will print its identifier to count the number of specific bugs triggered by the fuzzer. As shown in Table 2, there are four programs in LAVA-M dataset, uniq, base64, md5sum and who, respectively.

We list the number of real bugs contained in each program (line 1) and the number of bugs found by different fuzzers (line 2-4) in Table 2. As it shows, HDBFuzzer and QSYM both find more bugs than AFLGo-B on all the four programs. This is because many bugs in LAVA-M dataset are protected by magic bytes branches with computational features such as "5x+1==0xdeadbeaf", whose input needs to be computed. Compared with simple branches that don't need computation such as "x==0xdeadbeaf", such branches are difficult for AFLGo-B, while for QSYM and HDBFuzzer, it's much easier to solve the inputs satisfying such constraints. On the other hand, we can find in Table 2 that HDBFuzzer finds 665 more bugs than QSYM on who program, because HDBFuzzer can quickly identify difficult constraints and quickly solve such difficult constraints based on path abstraction.

Further, we randomly sample two bugs from each of the four programs in the LAVA-M dataset and set the target site to carry out directed fuzzing to evaluate the bug reproduction capability of different fuzzers on the LAVA-M dataset. The results are shown in Table 3, from which we can conclude that HDBFuzzer is superior than AFLGo-B and QSYM on all sampled bugs.

*4.2.2 Bug Reproduction and Target Reachability on Real-world Programs.* As stated before, three real-world programs that are widely used in embedded device firmware are selected: busybox, binutils and openssl, respectively. We list the version, CVE number, vulnerability type, vulnerability location and crash information of each CVE of the real-world programs in Table 4

We use the total success runs and $\mu$TTE (the average of TTEs collected from ten runs for each sample) to evaluate the bug reproduction capability of different fuzzers on real-world programs and $\mu$TTR (the average of TTRs collected from ten runs for each sample) to evaluate the target reachability of different fuzzers. As shown in Figure 4, HDBFuzzer is superior to AFLGo-B and QSYM in the total success runs and the total number of true crashing-inputs. By analyzing the total $\mu$TTE and the total $\mu$TTR of all the tests, we find that HDBFuzzer behaviors the best, followed by AFLGo-B and QSYM, QSYM behaviors the worst, which demonstrates the capability of the directness of HDBFuzzer.

## 5 CONCLUSION

In this paper, a target-oriented hybrid directed binary fuzzer (HDBFuzzer) is proposed to solve the vulnerability confirmation problem based on binary code similarity matching. Its core idea is to use DGF to carry out high-speed fuzzing test towards the target site(s), and at the same time adopts concolic execution to assist solving some complex constraints on the target path to the target site(s) to generate inputs that can pass through some specific complex branches for DGF. Compared to the traditional hybrid fuzzer, HDBFuzzer is a binary-oriented, target site(s)-driven, hybrid directed fuzzer, whose aim is to generate input that can reach the target site(s) as quickly as possible, so as to focus resources on fuzzing vulnerable code region to confirm whether there is a true vulnerability in the vulnerable code region. The results of the evaluation on the LAVA-M dataset and real-world programs related with 10 CVEs show that HDBFuzzer is superior to the state-of-the-art AFLGo-B and QSYM on the bugs founded, bug reproduction capability and target site reachability, which depends the ability of its directness

**Table 3: Bug reproduction of Different Fuzzers on LAVA-M Dataset**

| TTE(min)Program | AFLGo-B | | QSYM | | HDBFuzzer | |
|---|---|---|---|---|---|---|
| | bug1 | bug2 | bug1 | bug2 | bug1 | bug2 |
| **base64** | 32.43 | 30.56 | 21.54 | 28.23 | 18.51 | 20.45 |
| **md5sum** | 14.31 | 16.71 | 12.56 | 14.34 | 9.67 | 10.56 |
| **uniq** | 31.45 | 29.81 | 30.14 | 28.54 | 20.81 | 18.35 |
| **who** | 52.53 | 59.82 | 46.23 | 50.12 | 49.34 | 51.3 |

**Table 4: Real-world Programs and CVE Information**

| program | version | CVE | type | position | crash |
|---|---|---|---|---|---|
| binutils | 2.25.1 | CVE-2016-4489 | integer overflow | gnu_speical, libiberty/cplus-dem.c | ✓ |
| binutils | 2.25.1 | CVE-2016-4492 | integer overflow | do_type, libiberty/cplus-dem.c | ✓ |
| binutils | 2.32 | CVE-2019-14444 | integer overflow | apply_relocations(), readelf.c | ✓ |
| busybox | 1.21 | CVE-2016-6301 | deny of service | recv_and_process_client_pkt, networking/ntpd.c | ✓ |
| busybox | 1.27.2 | CVE-2017-15873 | integer overflow | get_next_block, archival/libarchive/decompress_bunzip2.c | ✓ |
| busybox | 1.27.2 | CVE-2015-9261 | buffer overflow | huft_build, archival/libarchive/decompress_gunzip.c | ✓ |
| openssl | 0.9.8 | CVE-2014-3508 | buffer overflow | OBJ_obj2txt, crypto/objects/obj_dat.c | ✓ |
| openssl | 1.0.1f | CVE-2015-0290 | deny of service | ssl3_write_bytes, s3_pkt.c | ✓ |
| openssl | 1.0.1f | CVE-2016-2180 | buffer overflow | TS_OBJ_print_bio, crypto/ts/ts_lib.c | ✓ |
| openssl | 1.0.1f | CVE-2016-2842 | buffer overflow | doapr_outch, crypto/bio/b_print.c | ✓ |

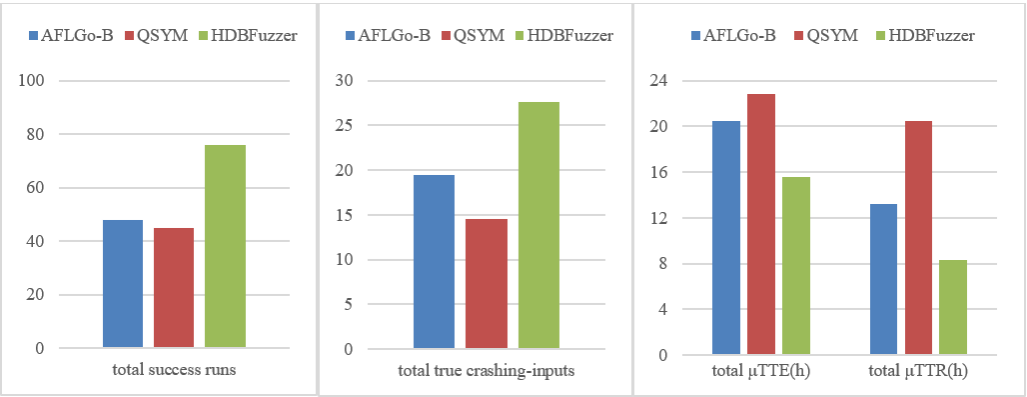


**Figure 4: Bug Reproduction and Target Reachability of Different Fuzzers on Real-world Programs.**

and its efficient constraint solving based on path abstraction. However, HDBFuzzer only supports x86 binaries now. In the future, we will improve HDBFuzzer to support binaries in ARM and mips.

## REFERENCES


[1] Yu Z, Cao R, Tang Q, *et al.* (2020). Order matters: Semantic-aware neural networks for binary code similarity detection[C]//Proceedings of the AAAI Conference on Artificial Intelligence. 34(01): 1145-1152.

[2] Liu B, Huo W, Zhang C, *et al.* (2018). $\alpha$diff: cross-version binary code similarity detection with dnn[C]//Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering. 667-678.

[3] Massarelli L, Di Luna G A, Petroni F, *et al.* (2019). Safe: Self-attentive function embeddings for binary similarity[C]//International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment. Springer, Cham, 309-329.

[4] Duan Y, Li X, Wang J, *et al.* (2020). Deepbindiff: Learning program-wide code representations for binary diffing[C]//Network and Distributed System Security Symposium.

[5] Sutton M, Greene A, Amini P (2007). Fuzzing: brute force vulnerability discovery[M]. Pearson Education.

[6] Böhme M, Pham V T, Nguyen M D, *et al.* (2017). Directed greybox fuzzing[C]//Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. 2329-2344.

[7] Chen H, Xue Y, Li Y, *et al.* (2018). Hawkeye: Towards a desired directed grey-box fuzzer[C]//Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. 2095-2108.

[8] Stephens N, Grosen J, Salls C, *et al.* (2016). Driller: Augmenting fuzzing through selective symbolic execution[C]//NDSS. 16(2016): 1-16.

[9] Yun I, Lee S, Xu M, *et al.* (2018). {QSYM}: A practical concolic execution engine tailored for hybrid fuzzing[C]//27th {USENIX} Security Symposium ({USENIX} Security 18). 745-761.

[10] Barrett C, Tinelli C (2018). Satisfiability modulo theories[M]//Handbook of model checking. Springer, Cham, 305-343.

[11] Österlund S, Razavi K, Bos H, *et al.* (2020). Parmesan: Sanitizer-guided greybox fuzzing[C]//29th {USENIX} Security Symposium ({USENIX} Security 20). 2289-2306.

[12] Zong P, Lv T, Wang D, *et al.* (2020). Fuzzguard: Filtering out unreachable inputs in directed grey-box fuzzing through deep learning[C]//29th {USENIX} Security Symposium ({USENIX} Security 20). 2255-2269.
[13] Zhao L, Cao P, Duan Y, *et al.* (2020). Probabilistic Path Prioritization for Hybrid Fuzzing[J]. IEEE Transactions on Dependable and Secure Computing, PP (99):1-1.
[14] Klees G, Ruef A, Cooper B, *et al.* (2018). Evaluating fuzz testing[C]//Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. 2123-2138.
[15] Dolan-Gavitt B, Hulin P, Kirda E, *et al.* (2016). Lava: Large-scale automated vulnerability addition[C]//2016 IEEE Symposium on Security and Privacy (SP). IEEE, 110-121.