Hawkeye: Towards a Desired Directed Grey-box Fuzzer

Hongxu Chen
Nanyang Technological University
Singapore, Singapore
hchen017@e.ntu.edu.sg

Yinxing Xue*
University of Science and Technology of China, Hefei, China
yxxue@ustc.edu.cn

Yuekang Li
Nanyang Technological University
Singapore, Singapore
yli044@e.ntu.edu.sg

Bihuan Chen
Fudan University
Shanghai, China
bhchen@fudan.edu.cn

Xiaofei Xie
Nanyang Technological University
Singapore, Singapore
xfxie@ntu.edu.sg

Xiuheng Wu
Nanyang Technological University
Singapore, Singapore
wuxh@ntu.edu.sg

Yang Liu
Nanyang Technological University
Singapore, Singapore
yangliu@ntu.edu.sg

ABSTRACT

Grey-box fuzzing is a practically effective approach to test real-world programs. However, most existing grey-box fuzzers lack directedness, i.e. the capability of executing towards user-specified target sites in the program. To emphasize existing challenges in directed fuzzing, we propose Hawkeye to feature four desired properties of directed grey-box fuzzers. Owing to a novel static analysis on the program under test and the target sites, Hawkeye precisely collects the information such as the call graph, function and basic block level distances to the targets. During fuzzing, Hawkeye evaluates exercised seeds based on both static information and the execution traces to generate the dynamic metrics, which are then used for seed prioritization, power scheduling and adaptive mutating. These strategies help Hawkeye to achieve better directedness and gravitate towards the target sites. We implemented Hawkeye as a fuzzing framework and evaluated it on various real-world programs under different scenarios. The experimental results showed that Hawkeye can reach the target sites and reproduce the crashes much faster than state-of-the-art grey-box fuzzers such as AFL and AFLGo. Specially, Hawkeye can reduce the time to exposure for certain vulnerabilities from about 3.5 hours to 0.5 hour. By now, Hawkeye has detected more than 41 previously unknown crashes in projects such as Oniguruma, MJS with the target sites provided by vulnerability prediction tools; all these crashes are confirmed and 15 of them have been assigned CVE IDs.

CCS CONCEPTS

• **Security and privacy → Vulnerability scanners**;

KEYWORDS

Fuzz Testing; Static Analysis

ACM Reference Format:
Hongxu Chen, Yinxing Xue, Yuekang Li, Bihuan Chen, Xiaofei Xie, Xiuheng Wu, and Yang Liu. 2018. Hawkeye: Towards a Desired Directed Grey-box Fuzzer. In *2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18), October 15–19, 2018, Toronto, ON, Canada.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3243734.3243849

*Corresponding author.

CCS '18, October 15–19, 2018, Toronto, ON, Canada

ACM ISBN 978-1-4503-5693-0/18/10…$15.00
https://doi.org/10.1145/3243734.3243849

1 INTRODUCTION

Security testing is one of the most effective vulnerability detection techniques for modern software. Among the security testing techniques, fuzzing [30], or fuzz testing, is regarded as the most effective and scalable, which provides various inputs to the program under test (PUT) and monitors for abnormal behaviors (e.g., stack or buffer overflow, invalid read/write, assertion failures, or memory leaks) [13]. Since the proposal, fuzzing has gained the popularity in industry and academia, and evolved into different types of fuzzers for different testing scenarios. Fuzzers can be classified as black-box, white-box or grey-box [10], according to their awareness of the internal structure of the PUT. Recently, grey-box fuzzers have been widely-used and proven to be effective [7]. Specificaly, AFL [48] and its derivations [6, 7, 12, 15, 24, 43] receive plenty of attentions.

In general, the existing grey-boxing fuzzers (GFs) aim to cover as many program states as possible within a limited time budget. However, there exist several testing scenarios in which only particular program states are concerned and required to be sufficiently tested. For example, if *MJS* [39](a JavaScript engine for embedded devices) has a vulnerability discovered on MSP432 ARM platform, similar vulnerabilities may occur in the corresponding code for the other platforms. In such a situation, the fuzzer should be *directed* to reproduce the bug at these locations. Another case is, when a bug is patched, the programmers need to check whether the patch completely fixes the bug. This requires the fuzzer to focus its efforts on those patched code. In both scenarios, the fuzzer is required to be directed to reach certain user specified locations in the PUT. For clarity, we name such locations as *target sites*. Following the

definition in [6], we name the fuzzers that can fulfil the directed fuzzing task as *directed fuzzers*.

As the state-of-the-art directed grey-box fuzzer (DGF for short), AFLGo [6] casts the reachability of target sites as an *optimization problem* and adopts a meta-heuristic to promote the test seeds with shorter distances. Here, the distance is calculated according to the average *weight* of basic blocks on the input seed's execution trace to the target basic block(s), where the weight is determined by the edges in the call graph and control flow graphs of the program, and the meta-heuristic is simulated annealing [22]. Based on these, AFLGo solves the *power scheduling problem* for directed fuzzing — how many new inputs (termed "energy" in AFLGo) should be generated from the current test seed. To summarize, represented by AFLGo, DGF achieves the goal of reaching the target sites by *combing both static analysis and dynamic analysis*.

A pure dynamic execution can only get the feedback based on the traces it has *already* covered without any awareness about the predefined target sites. Thus, ***static analysis*** is required to extract the necessary information for guiding the execution towards the target sites for DGFs. The most widely used approach is to calculate the *distance* (or weight) to the target sites for the components (e.g., basic blocks, functions) of the PUT, so that when executed, DGFs can judge the *affinity* between current seed and the target sites from the components in the execution traces. The major challenge is that the distance needs to be effectively calculated without compromising certain desired features. Particularly, it should help to retain the seed diversity [4]. For example, the existing seed distance calculation algorithm used in AFLGo always favors shortest path that leads to the targets (see § 2.1), which may starve inputs that could be more easily mutated to reach the target site and further trigger crashes. The author of libFuzzer [26] argues that not taking into account *all possible traces* may fail to expose the bugs hidden deeply in longer paths [37]. This derives the first desired property **P1**. Another challenge is that the static analysis should provide precise information with acceptable overheads. This is because that coarse static analyses will not benefit the dynamic fuzzing much, while heavyweight static analyses themselves may take considerable time before the dynamic fuzzing starts. This challenge derives the second desired property **P2**. Hence, the first problem is *to have a proper static analysis which can collect necessary information for DGF*.

After extracting the information with static analysis, there are several challenges in ***dynamic analysis*** — how to dynamically adjust different strategies for the purpose of reaching the target sites as fast as possible. The first challenge is how to properly *allocate* energy to the inputs with different distances and how to *prioritize* the inputs closer to the targets. This derives the third desired property **P3**. The second challenge is how to adaptively change the mutation strategies, since GFs may possess various mutation operators at both coarse-grained (e.g., bulk deletion) and fine-grained (e.g., bitwise flip) levels. This derives the fourth desired property **P4**. Hence, the second problem is *to make proper adjustments for the dynamic strategies used in DGF*.

To emphasize the two aforementioned problems, an ideal DGF is expected to hold the following desired properties (§2.2):

- **P1** The DGF should have a *robust* distance-based mechanism that can guide the directed fuzzing by considering all traces to the targets and avoiding the bias to certain traces.
- **P2** The DGF should strike a *balance* between overheads and utilities in static analysis.
- **P3** The DGF should prioritize and schedule the seeds to reach target sites *rapidly*.
- **P4** The DGF should adopt an *adaptive* mutation strategy when the seeds cover different program states.

In this paper, we propose our solutions to achieve the four desired properties for DGF. For **P1**, we propose to apply the static analysis results to augment the adjacent-function distance (§4.2); and the function level distance and basic block level distance should be calculated based on the augmented adjacent-function distance to simulate the affinities between functions (§4.3). Meanwhile, during fuzzing, we calculate *basic block trace distance* and *covered function similarity* of the execution trace to that of the target functions (§4.4) by integrating the static analysis results with the runtime execution information. For **P2**, we propose to apply the analysis based on call graph (CG) and control flow graph (CFG), i.e., the function level reachability analysis, the points-to analysis for function pointers (indirect calls), and the basic block metrics (§4.1). For **P3**, we propose to combine the basic block trace distance and covered function similarity for solving the power scheduling problem (§4.4) and the seed priorization problem (§4.6). For **P4**, we propose to apply an adaptive mutation strategy according to the reachability analysis and covered function similarity (§4.5).

By taking these properties into account, we implemented our DGF, Hawkeye, and conducted a thorough evaluation with various real-world programs. The experimental results show that in most cases, Hawkeye outperforms the state-of-the-art grey-box fuzzers in terms of the time to reach the target sites and the time to expose the crashes. Particularly, Hawkeye can expose certain crashes up to 7 times faster than the state-of-the-art AFLGo, reducing the time to exposure from 3.5 hours to 0.5 hours.

In practice, Hawkeye has been successfully discovering crashes with the suspicious target sites reported by other vulnerability detection tools and successfully found more than 41 previously unknown crashes in projects Oniguruma [23], MJS[39], etc. All these vulnerabilities have been confirmed and fixed; among them, 15 vulnerabilities have been assigned unique CVE IDs.

The main contributions of this paper are summarized as follows:

(1) We analyzed the challenges in directed grey-box fuzzing and summarized the four desired properties for DGFs.
(2) We provided a measure of power function that can guide the fuzzer towards the target sites effectively.
(3) We proposed a novel approach to boost the convergence speed to the target sites by utilizing power scheduling, adaptive mutation and seed prioritization.
(4) We implemented a novel fuzzing framework that organically combines these ideas and thoroughly evaluated our results in both crash reproduction and target site covering.

2 DESIRED PROPERTIES OF DGF

In this section, we first show an example to illustrate the difficulties in DGF. Based on the observations from the example, we then

Functions in a Crashing Trace	File & Line	Symbol
main	nm.c :1794	M
...	...	...
_bfd_dwarf2_find_nearest_line	dwarf2.c :4798	a
comp_unit_find_line	dwarf2.c :3686	b
comp_unit_maybe_decode_line_info	dwarf2.c :3651	c
decode_line_info	dwarf2.c :2265	d
concat_filename	dwarf2.c :1601	T
...	...	Z
Functions in a Normal Trace	**File & Line**	**Symbol**
main	nm.c :1794	M
...	...	...
_bfd_dwarf2_find_nearest_line	dwarf2.c :4798	a
scan_unit_for_symbols	dwarf2.c :3211	e
concat_filename	dwarf2.c :1601	T
...	...	Z

Figure 1: Two execution traces related to CVE-2017-15939: M is the main function, T is the target function, Z is the exit.

propose four desired properties for an ideal DGF. Finally, we review the state-of-the-art DGF, namely AFLGo [6], with respect to these four desired properties.

2.1 Motivating Example

Fig. 1 shows two execution traces related to CVE-2017-15939 [36], which is a NULL pointer dereference bug caused by an incomplete fix in CVE-2017-15023 [35]. This vulnerability is difficult for the existing GFs to discover. For instance, AFL [48] fails to detect this vulnerability within 24 hours in all the 10 different runs we conducted. This bug is triggered in `nm` from GNU binutils. In function `concat_filename`, a NULL pointer is assigned and used without checking, which triggers the segmentation fault. From a patch testing perspective, we would like to target `concat_filename` (subsequently, we will denote this as T) and guide the fuzzing to reproduce the crashing trace (i.e., $\langle a, b, c, d, T, Z\rangle$ in Fig. 2).

For simplicity, in Fig. 2, we illustrate only three representative traces for the CVE-2017-15939 by omitting 1) the overlapping functions before a and 2) the other traces that do not pass the target function T. The difficulty in discovering this CVE for the general GFs (e.g., AFL) arises from the fact that the target function T is deeply hidden in the crashing trace. As shown in Fig. 2, the call chain of $\langle a, e, T, Z\rangle$ is shorter than $\langle a, b, c, d, T, Z\rangle$.

Since most of the GFs (such as AFL, LibFuzzer, etc.) are supposed to be *coverage oriented*, and do not care specially about the targets, they may not put most of their efforts in generating test seeds that reach function T and testing the function throughly. For DGFs, although there suppose to be some efforts to guide the fuzzing procedure to favor *some* traces leading to T and focus more on these traces, they may frequently miss *all* the traces. For example, if AFLGo detects that two traces can reach the target sites, it will highly likely favors the trace with shorter path: the distance between the seed to the target is determined by the average distance of the components (basic blocks or functions) in the execution trace to the targets, where the components' distance to the target sites are essentially determined by the number of edges between the components to the targets. This mechanism causes AFLGo to give more energy to the trace $\langle a, e, T, Z\rangle$ since it reaches the target T *and* the induced trace distance is smaller than $\langle a, b, c, d, T, Z\rangle$; on the other hand, less attention is put on $\langle a, b, c, d, T, Z\rangle$, which however causes the crash under some circumstances. Worse still, other traces like $\langle a, e, f, Z\rangle$ may be mistakenly assigned with more energy. As a result, AFLGo was also not able to reproduce the crash within 24 hours in any of the 10 runs we conducted.

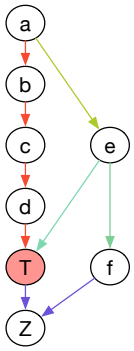

Figure 2: The fuzzing scenario modeled from Fig. 1: $\langle a, b, c, d, T, Z\rangle$ is a crashing trace passing T, $\langle a, e, T, Z\rangle$ is a normal trace passing T, and $\langle a, e, f, Z\rangle$ is a trace not passing T.

The challenges of the existing DGF roots in the following aspects: 1) the target functions may appear in several places in PUT, and multiple different traces may lead to the target. 2) since the call graph majorly affects the calculation of the trace distance (the dissimilarity with the target sites), it needs to be accurately built; in particular, the indirect calls among functions should not be ignored. If the above two issues are not well handled, the distance-based guiding mechanism for DGF will get hindered and fail in such cases.

2.2 Desired Properties of Directed Fuzzing

As observed from the above example, an ideal DGF should possess the following desired properties.

2.2.1 ***P1****. The DGF should define a* ***robust*** *distance-based mechanism that can guide the directed fuzzing by avoiding the bias to some traces and considering all traces to the targets.* Different from general GFs, to reach the targets, there may exist several execution traces towards the targets. More often than not, a target function could appear several times in the code and be called even from different entries of the code. Without any static information as guidance, during the fuzzing process, the fuzzer knows nothing about the execution traces that can cover the targets before the targets have been executed; and even if the targets have already been covered, the fuzzer does not know whether there are *other* traces that can lead to these targets. Hence, the guiding mechanism should help find all the traces that lead to the targets — taking Fig. 2 as an example, in the AFL fuzzing process, trace $\langle a, b, c, d, T, Z\rangle$ may not be ever exercised by the existing inputs due to the existence of a strong precondition before a. Hence, *the guiding mechanism could provide the knowledge of all possible traces leading to targets and guide the mutation towards it via gradually reducing the distance.*

However, for DGF, awareness of all possible traces towards the targets is not enough: the distance to the targets for all traces should be properly calculated so that all traces reachable to the targets will be assigned more energy compared to other traces. For Fig. 2,

T is the target and we would like to check the functionality of T[1]. Intuitively, traces $\langle a, e, T, Z\rangle$ and $\langle a, b, c, d, T, Z\rangle$ should be treated without bias as both of them can lead to the target site, while $\langle a, e, f, Z\rangle$ should be less important as it misses the target site.

2.2.2 ***P2****. The DGF should strike a* ***balance*** *between overheads and utilities in static analysis.* Effective static analysis can benefit the dynamic fuzzing procedure in two aspects: 1) In real world C/C++ programs, there are indirect function calls (e.g., passing a function pointer as a parameter in C, or using function objects and pointers to member functions in C++). In the presence of indirect calls, call sites cannot be observed directly from the source code or binary instructions. So the trade-offs between overheads and utilities need to be made for analyzing them. 2) Not all call relations should be treated equally. For example, some functions occur multiple times in its calling functions, implying that they have higher chance to be called at runtime. From the static analysis perspective, we need to provide a way to distinguish these scenarios. As to function level distances between functions that have *immediate* calling relations, it is intuitive that callees called in multiple times in different branches should be "closer" to the caller.

To sum up, taking Fig. 2 as an example, *the desired design* for the DGF is: 1) if function a (transitively) calls T in an indirect way (i.e., one or more calls in the chain $a \rightarrow b \rightarrow c \rightarrow d \rightarrow T$ are through function pointers), the static analysis should capture such indirect calls, otherwise the distance from a to T will be not available (i.e., treated as unreachable). 2) if the callee appears in more different branches and occurs more times in its caller, a smaller distance should be given since it may have more chance of being called for reaching the target(s). However, modeling the *actual* branch conditions in static phrase is impractical due to the inherent limitations of static analysis. For example, given a nontrivial code segment, it is hard to predict whether the true branch of a predicate will be executed more often than its false branch during runtime. On the other hand, tracking symbolic conditions *dynamically* would be too time costly in a grey-box fuzzing setting.

2.2.3 ***P3****. The DGF should select and schedule the seeds to reach target sites* ***rapidly****.* AFL determines how many new inputs should be generated (i.e., "energy") from a test seed to improve the fuzzing effectiveness (i.e., increase the coverage); this is termed "power scheduling" in [6, 7]. In directed fuzzing, the goal of fuzzing is not to reach *the upper-limit of coverage* as fast as possible, but reach the *particular targets* as fast as possible. Hence, power scheduling in DGF should determine how many new inputs should be generated from a test seed in order to get a new mutated input that leads to the target sites [6]. Similarly, the seed prioritization in DGF is to determine an optimized fuzzing order of test seeds to reach target sites as fast as possible. Both of them can be guided by the distance-based mechanism which measures the affinity between the current seed to the target sites.

For *power scheduling*, the desired design is that the seed trace with a smaller distance to targets should be assigned more energy for fuzzing, as the trace closer to the target sites gets better chance to reach there. Therefore, $\langle a, e, T, Z\rangle$ should be allocated with similar energy with $\langle a, b, c, d, T, Z\rangle$, and $\langle a, e, f, Z\rangle$ should have less energy than the previous two. For *seed prioritization*, seeds that have smaller distance ("closer") to the targets should be fuzzed earlier in subsequent mutations. Therefore, $\langle a, e, T, Z\rangle$ and $\langle a, b, c, d, T, Z\rangle$ should be put ahead of $\langle a, e, f, Z\rangle$.

2.2.4 ***P4****. The DGF should adopt an* ***adaptive*** *mutation strategy when the seeds cover the different program states.* GFs usually apply different mutations, such as bitwise flip, byte rewrite, chunk replacement, to generate new test seeds from the existing one. In general, these mutators can be categorized into two levels: fine-grained mutations (e.g., bitwise flip) and coarse-grained mutations (e.g., chunk replacement). Although there is no direct evidence that fine-grained mutations will likely preserve the execution traces, it is widely accepted that a coarse-grained random mutation has a high chance to change the execution trace greatly. Therefore, the desired design is that when a seed has already reached the target sites (including target lines, basic blocks or functions), it should be given less chances for coarse-grained mutations.

For the example in Fig. 2, consider the case where the DGF has already reached the target function via trace $\langle a, b, c, d, T, Z\rangle$, but crash is not triggered yet. Now, the DGF should allocate less chances for coarse-grained mutations for the input of $\langle a, b, c, d, T, Z\rangle$. Meanwhile, if DGF has just started up and $\langle a, b, c, d, T, Z\rangle$ has not been reached yet, then the DGF should give more chances for coarse-grained mutations.

2.3 AFLGo's Solution

In this section, we evaluate the solution of AFLGo against the four desired properties to demonstrate the significances of these properties as well as some useful concepts in DGF.

For P1. For Fig. 2, based on the distance formula defined in AFLGo the trace distances are: $d_s(abcdTZ) = (2+3+2+1+0)/5 = 1.6$, $d_s(aeTZ) = (2+1+0)/3 = 1$ and $d_s(aefZ) = (2+1)/2 = 1.5$ [2]. Given these three execution traces, the energy assigned to them will be $\langle a, e, T, Z\rangle > \langle a, e, f, Z\rangle > \langle a, b, c, d, T, Z\rangle$. This is problematic: the normal trace $\langle a, e, T, Z\rangle$ is overemphasized; the crashing trace $\langle a, b, c, d, T, Z\rangle$ is however considered the least important, even less important than the trace $\langle a, e, f, Z\rangle$ that fails to reach the target T.

For P2. AFLGo only considers the explicit call graph information. As a result, all function pointers are treated as *external nodes* which are ignored during distance calculation. This means that, in an extreme case, if the target function is called via a function pointer, its distance from the actual caller is undefined. For example, in Fig. 2, if d and e call T via function pointers, both d and e will be mistakenly considered unreachable to T; consequently, all nodes except for T will be considered unreachable to T. Therefore essentially there is no directedness in such a case.

Besides, AFLGo counts the same callee in its callers only once, and it does not differentiate multiple call patterns between the caller and callee (see §4.2). The function level distance is calculated on the call graph with the Dijkstra shortest path, assuming the weight of two adjacent nodes (functions) in the call graph always to be 1, which will distort the distance calculation.

[1] If we know previously that *only* the traces that involve d and T may cause crashes, we can set *both* d and T as the target sites.

[2] In fact, AFLGo calculates the trace distance at the *basic block* level with harmonic mean of the accumulative distance; nevertheless, the essential idea is the same.

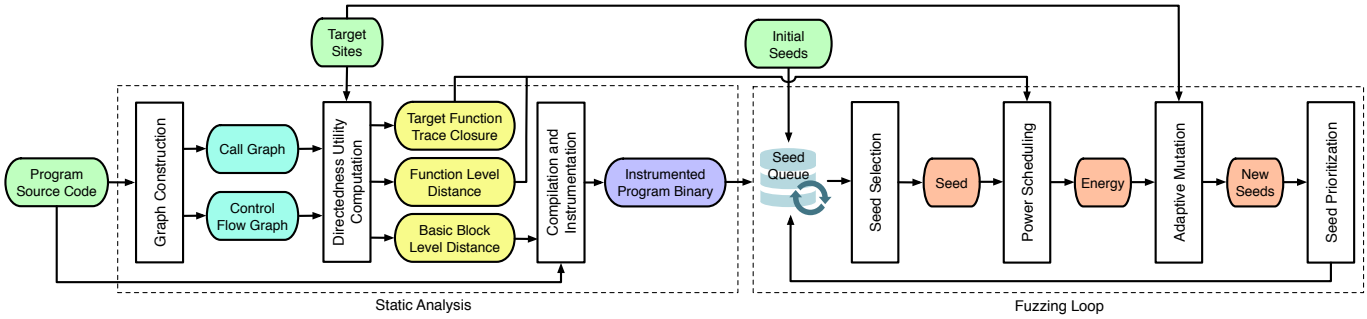

Figure 3: Approach Overview of Hawkeye

For P3. AFLGo applies a simulated annealing based power scheduler: it favors those seeds that are closer to the targets by assigning more energy to them for mutation; the applied cooling schedule initially assigns smaller weight on the effect of "distance guidance", until it reaches the "exploitation" phrase. It solves the "exploration vs exploitation" problem [8] and mitigates the imprecision issue brought by the statically calculated basic block level distance. In our opinion, this is an effective strategy. The problem is that there is no prioritization procedure so the newly generated seeds with smaller distance may wait for a long to be mutated.

For P4. The mutation operators of AFLGo come from AFL's two non-deterministic strategies: 1) *havoc*, which does purely randomly mutations such as bit flips, chunk replacement, etc; 2) *splice*, which generates seeds from some random byte parts of two existing seeds. Notably, during runtime AFLGo excludes all the deterministic mutation procedures and relies purely on the power scheduling on havoc/splice strategies. The randomness of these two strategies can indeed favor those with smaller distances to the targets. However, it may also destroy the existing seeds that are close to the targets. In fact, some subtle vulnerabilities can only be reached with some special preconditions. In reality, an incomplete fix may still leave some concern cases to be vulnerable; for example, CVE-2017-15939 is caused by an incomplete fix for CVE-2017-15023. Hence, AFLGo lacks the adaptive mutation strategies, which will mutate arbitrarily even when the current seeds are close to the targets enough.

Summary. Taking AFLGo as example, we can summarize the following suggestions to improve DGFs:

(1) For **P1**, a more accurate distance definition is needed to retain trace diversity, avoiding the focus on short traces.
(2) For **P2**, both direct and indirect calls need to be analyzed; various call patterns need to be distinguished during static distance calculation.
(3) For **P3**, a moderation to the current power scheduling is required. The distance-guided seed prioritization is also needed.
(4) For **P4**, the DGF needs an adaptive mutation strategy, which optimally applies the fine-grained and coarse-grained mutations when the distance between the seed to the targets is different.

3 APPROACH OVERVIEW

In this section, we briefly introduce the workflow of our proposed approach, named Hawkeye. An overview of Hawkeye is given in Fig. 3, which consists of two major components, i.e., *static analysis* and *fuzzing loop*.

3.1 Static Analysis

The inputs of static analysis are the *program source code* and the *target sites* (i.e., the lines of code that the fuzzer is directed to reach). We derive the basic blocks and functions where the target sites reside in, and call them *target basic blocks* and *target functions*, respectively. The main output of static analysis is the *instrumented program binary* with the information of *basic block level distance*.

First, we precisely construct the call graph (CG) of the target program based on the inclusion-based pointer analysis [3] to include all possible calls. Besides, for each function, we construct the control flow graph (CFG) (§4.1).

Second, we compute several utilities that are used to facilitate the directedness in Hawkeye based on CG and CFG (§4.3).

(1) **Function level distance** is computed based on CG by augmenting adjacent-function distance (§4.2). This distance is utilized to calculate the *basic block level distance*. It is also used during the fuzzing loop to calculate the *covered function similarity* (§4.4).
(2) **Basic block level distance** is computed based on the function level distance, together with the CG and the functions' CFGs. This distance is statically instrumented for each basic block that is considered to be able to reach one of the target sites. During the fuzzing loop, it is also used to calculate the *basic block trace distance* (§4.4).
(3) **Target function trace closure** is computed for each target site according to the CG to obtain the functions that can reach the target sites. It is used during the fuzzing loop to calculate the *covered function similarity* (§4.4).

Finally, the target program is instrumented to keep track of the edge transitions (similar to AFL), the accumulated basic block trace distance (similar to AFLGo), and the covered functions.

3.2 Fuzzing Loop

The inputs of fuzzing loop are the *instrumented program binary*, the *initial test seeds*, the *target sites* as well as the information of *function level distance* and *target function trace closure*. The outputs of fuzzing loop are the test seeds that cause abnormal program behaviors such as crashes or timeouts.

During fuzzing, the fuzzer selects a seed from a priority seed queue. The fuzzer applies a *power scheduling* against the seed with the goal of giving those seeds that are considered to be "closer"

to the target sites more mutation chances, i.e., energy (§4.4). Specifically, this is achieved through a power function, which is a combination of the *covered function similarity* and the *basic block trace distance*. For each newly generated test seed during mutation, after capturing its execution trace, the fuzzer will calculate the covered function similarity and the basic block trace distance based on the utilities (§3.1). For each input execution trace, its basic block trace distance is calculated as the accumulated basic block level distances divided by the total number of executed basic blocks; and its covered function similarity is calculated based on the overlapping of current executed functions and the target function trace closure, as well as the function level distance.

After the energy is determined, the fuzzer adaptively allocates mutation budgets on two different categories of mutations according to mutators' granularities on the seed (§4.5). Afterwards, the fuzzer evaluates the newly generated seeds to prioritize those that have more energy or that have reached the target functions (§4.6).

4 METHODOLOGY

In this section, we will elaborate the key components in Fig. 3 featuring the four desired properties.

4.1 Graph Construction

To calculate the accurate distance from a test seed to the oracle seed executing the target sites, we first build up the CG and CFG, then combine them to construct the final inter-procedural CFG. Note that CG is used to compute the function level distance in §4.2 and §4.3, CFG together with CG (i.e., inter-procedural CFG) is used to compute the basic block distance in §4.3.

To identify the indirect call in call graph, we propose to apply the inclusion-based pointer analysis [3] against the function pointers of the whole program. The core idea of this algorithm is to translate the input program with statements of the form $p := q$ to constraints of the form "q's points-to set is a subset of p's points-to set". Essentially, the propagation of the points-to set is applied with four rules namely *address-of*, *copy*, *assign*, *dereference*. This analysis is context-insensitive and flow-insensitive, meaning that it ignores both the calling context of the analyzed functions and the statement ordering inside functions, and eventually only computes a single points-to solution that holds for all the program points. Usually, a fixed point of the points-to sets will be reached at the end of the analysis. Among these, points-to sets of the function pointers inside the whole program are calculated, resulting in a relatively precise call graph including all the possible direct and indirect calls. The complexity of this pointer analysis is $\Theta(n^3)$. The reason that we do not apply context-sensitive or flow-sensitive analyses lies in the fact that they are computationally costly and not scalable to large projects. Despite that, our call graph is still much more precise than the one generated by LLVM's builtin APIs, which does not contain any explicit nodes that represent indirect calls.

The control flow graph of each function is generated based on LLVM's IR. The inter-procedure flow graph is constructed by collecting the call sites in all the CFGs and the CG of the whole program. By applying these static analyses, we achieve **P2**.

```
void fa(int i) {
  if (i > 0) {
    fb(i);
  } else {
    fb(i*2);
    fc();
  }
}
```

(a)

```
void fa(int i) {
  if (i > 0) {
    fb(i);
    fb(i*2);
  } else {
    fc();
  }
}
```

(b)

Figure 4: An example illustrating different call patterns

4.2 Adjacent-Function Distance Augmentation

To achieve **P1**, we propose to implement a lightweight static analysis that considers the patterns of the (immediate) call relation based on the generated call graph. As discussed in §2.2.2, under different context, the distances from the calling function to the immediately called function may not be exactly the same. Given functions f_a, f_b, f_c, there may exist several different call patterns in the call graph. For example, in Fig. 4a and Fig. 4b, there are calls $f_a \rightarrow f_b$ and $f_a \rightarrow f_c$ in both cases. However, in Fig. 4a f_a is bound to call f_b (since f_b appears in both *if* and *else* branches in f_a), but not necessary to call f_c; in Fig. 4b, both f_b and f_c are not necessary to be called by f_a. From a probability perspective, we would think that in both cases the distance from f_a to f_b should be smaller than the distance from f_a to f_c, and the distance from f_a to f_b in Fig. 4a should be smaller than that in Fig. 4b.

Therefore, we propose two metrics to augment the distance that is defined by immediate calling relation between caller and callee.

(1) Call site occurrences C_N of a certain callee for a given caller. More occurrences of callee could incur more chance that callee will be dynamically executed with more different (actual) parameters, and in return the distance between the caller to the callee will be smaller. We apply a factor $\Phi(C_N) = \frac{\phi \cdot C_N + 1}{\phi \cdot C_N}$ to denote this effect, where ϕ is a constant value (usually, $\phi = 2$).
(2) The number of basic blocks C_B in the caller that contains at least one call site of the callee. The rationale is that, with more branches that have a call site, more different execution traces will include the callee. The factor function $\Psi(C_B) = \frac{\psi \cdot C_B + 1}{\psi \cdot C_B}$ denotes this effect, and ψ is a constant value (usually, $\psi = 2$).

Note that both factor functions are monotone decreasing functions; also, Φ converges to 1 when $C_N \rightarrow \infty$ and Ψ converges to 1 when $C_B \rightarrow 1$. Given a (direct or indirect) immediate function call pair (f_1, f_2) where f_1 is the caller and f_2 is the callee, the original distance between f_1 and f_2 is 1 (see AFLGo [6]). Now, with the two metrics mentioned above, we can define the augmented distance between the function pairs that holds an immediate call relation. The final adjustment factor will be a multiplication of Φ and Ψ, and the *augmented adjacent-function distance* is

$$d'_f(f_1, f_2) = \Psi(f_1, f_2) \cdot \Phi(f_1, f_2) \tag{1}$$

where $d'_f(f_1, f_2)$ refers to the *augmented direct* function distance.

As an example, in Fig. 4a, for f_b, $C_N(f_a, f_b) = 2$, $C_B(f_a, f_b) = 2$; and for f_c, $C_N(f_a, f_c) = 1$, $C_B(f_a, f_c) = 1$. Assume $\phi = 2$ and $\psi = 2$ and assume the original distance $d_f(f_a, f_b) = d_f(f_a, f_c) = 1$,

the augmented distances will be $d'_f(f_a, f_c) = \frac{3}{2} \cdot \frac{3}{2} = 2.25$, and $d'_f(f_a, f_b) = \frac{5}{4} \cdot \frac{5}{4} = 1.56$.

A special case not shown in the above examples is that some branches form cycles (i.e., loops). Indeed, these functions may be called multiple times at runtime. However, it is uncertain that *how many times* they will be executed across different runs when fed with different seeds. Fortunately, actual execution on one call site of a callee inside one loop typically has similar effect — the loop explores the similar program states and benefits less in covering new paths. Hence, the function call inside the loop does not bring many execution trace diversities like the scenario where the same callee occurs in multiples branches with significantly different parameters.

The applied approach aims to make a trade-off between the efficiency and the utility of the static analysis. Therefore, we do not consider the solution space of any branch condition that may affect the *runtime reachability* in the CFG. For example, in Fig. 4a, if we change the condition check *i>0* to be *i==0*, the true branch will be executed only when the input value of *i* is 0. It is tempting to assign a smaller distance to the code segments in the false branch. However, since the PUT is usually nontrivial, it is impractical to statically formulate the exact constraint set of the preconditions before reaching function f_a and predicate the branches's actual execution probabilities. One common scenario is that the branch condition *i==0* is used for checking the return status code of an external function call, at runtime it may actually execute the true branch more often than the false branch.

4.3 Directedness Utility Computation

In §4.2, the augmented function distance is calculated on two adjacent functions according to their call patterns. By assigning the adjacent-function distance as the weight of the edges in the call graph, we can calculate the function level distance for any two functions with the Dijkstra shortest path algorithm, beyond which we can further derive the basic block level distance. Besides, we also compute the target function trace closure which will be used to calculate the covered function similarity in §4.4.

Function Level Distance. This distance is calculated according to CG. It tells the (average) distance from the current function to target functions. Given a function *n*, its distance to the target function set T_f is defined as:

$$d_f(n, T_f) = \begin{cases} \textbf{undefined.} & \text{if } R(n, T_f) = \emptyset \\ [\Sigma_{t_f \in R(n,T_f)} d_f(n, t_f)^{-1}]^{-1} & otherwise \end{cases} \quad (2)$$

where $R(n, T_f) = \{t_f | \textbf{reachable}(n, t_f)\}$, which is the set of target functions that can be statically reached from *n* in CG, and $d_f(n, t_f)$ is the *dijkstra shortest path based on augmented function distance* from *n* to a given target function t_f in CG.

Basic Block Level Distance. Given a function *n* and two basic blocks m_1 and m_2 inside, the basic block level distance $d_b^o(m_1, m_2)$ is defined as the minimal number of edges from m_1 to m_2 in the CFG $G(n)$. The set of functions called inside basic block *m* is denoted as $C_f(m)$, then $C_f^T(m) = \{n | R(n, T_f) \neq \emptyset, n \in C_f(m)\}$, and $Trans_b = \{m | \exists G(n), m \in G(n), n \in F, C_f^T(m) \neq \emptyset\}$, where *F* is the set of all functions. Given a basic block *m*, its distance to the target basic blocks T_b are defined as:

$$d_b(m, T_b) = \begin{cases} 0 & \textbf{if } m \in T_b \\ c \cdot \min_{n \in C_f^T(m)}(d_f(n, T_f)) & \textbf{if } m \in Trans_b \\ [\Sigma_{t \in Trans_b}(d_b^o(m, t) + d_b(t, T_b))^{-1}]^{-1} & otherwise \end{cases} \quad (3)$$

where *c* is a constant that magnifies function level distance.

Note that Equation 2 and 3, on their own, are the same as those in AFLGo [6]. However, $d_f(n, t_f)$ for these equations in AFLGo is simply the Dijkstra shortest distance on a CG where the weight of edges (i.e., adjacent function distance) is 1.

Target Function Trace Closure. This utility, $\xi_f(T_f)$, is calculated by collecting all the predecessors that can statically lead to the target functions T_f, until the entry function *main* has been reached. We choose *not* to exclude those that are *considered* unreachable from entry function due to the limitations of static analysis. In the example in Fig. 2, $\xi_f(T_f) = \{a, b, c, d, e, T\}$.

4.4 Power Scheduling

During dynamic fuzzing, we apply power scheduling on a selected seed based on two dynamically-computed metrics: basic block trace distance and target function trace similarity.

Basic Block Trace Distance. The distance between the seed *s* to the target basic blocks T_b is defined as:

$$d_s(s, T_b) = \frac{\Sigma_{m \in \xi_b(s)} d_b(m, T_b)}{|\xi_b(s)|} \quad (4)$$

where $\xi_b(s)$ is the execution trace of a seed *s* and contains all the basic blocks that are executed. Hence, the basic idea of Equation 4 is that: for all the basic blocks in the execution trace of *s*, we calculate the average basic block level distance to the target basic blocks T_b. Note that Equation 4 is also the same as the one in AFLGo [6].

It then applies a feature scaling normalization to get the final distance $\tilde{d}_s(s, T_b) = \frac{d_s(s,T_b) - minD}{maxD - minD}$ where *minD* (or *maxD*) is the smallest (or largest) distances ever met.

Covered Function Similarity. This metric measures the similarity between the execution trace of the seed and the target execution trace *on the function level*. We do not track the basic block level trace similarity since that would introduce considerable overheads. The similarity is calculated based on the intuition that seeds covering more functions in the "expected traces" will have more chances to be mutated to reach the targets. This similarity is calculated by tracking the function *sets* the current seed covered (denoted as $\xi_f(s)$) and comparing it with the target function trace closure $\xi_f(T_f)$. In the example in Fig. 2, $\xi_f(abcdTZ) = \{a, b, c, d, T\}$, $\xi_f(aeTZ) = \{a, e, T\}$ and $\xi_f(aefZ) = \{a, e\}$.

The covered function similarity is then determined by the following formula:

$$c_s(s, T_f) = \frac{\Sigma_{f \in \xi_f(s) \cap \xi_f(T_f)} d_f(f, T_f)^{-1}}{|\xi_f(s) \cup \xi_f(T_f)|} \quad (5)$$

$d_f(f, T_f)$ is the function level distance calculated with Equation 2. Similar to d_s, a feature scaling normalization is also applied and the final similarity is denoted as $\tilde{c}_s$. Note that this similarity metric is uniquely proposed in our approach.

Scheduling. Scheduling deals with the problem how many mutation chances will be assigned to the given seed. The intuition is that if the trace that the current seed executes is "closer" to any of the expected traces that can reach the target site in the program, more mutations on that seed should be more beneficial for generating expected seeds. A scheduling purely based on *trace distance* may favor certain patterns of traces. For AFLGo, as mentioned in §2.2.2, the shorter paths will be assigned more energy, which may starve longer paths that are still reachable to the target sites. To *mitigate* this, we propose the *power function* that considers both *trace distance* (based on basic block level distance) and *trace similarities* (based on covered function similarity):

$$p(s, T_b) = \tilde{c}_s(s, T_f) \cdot (1 - \tilde{d}_s(s, T_b)) \tag{6}$$

It is obvious that the value of $p(s, T_b)$ fits into $[0, 1]$ since both the multipliers are in $[0, 1]$.

Compared to AFLGo's approach, which only considers basic block trace distance (d_s, or $\tilde{d}_s$), our power function balances the effect of shorter paths and the longer paths that can reach the target. Logically, there are some differences between c_s and d_s:

(1) d_s considers both the effects of CG and the CFGs; c_s considers only the effects of CG. For d_s, the major effect is still CG due to the magnification factor c used in Equation 2.
(2) d_s does not penalize traces that do not lead to the targets, while c_s penalizes them via a union of $\xi_f(s)$ (by tracking function level traces) and $\xi_f(T_f)$.
(3) Given multiple traces that can lead to the targets, d_s favors those that have short lengths, but c_s favors those with longer lengths of common functions in expected trace.

In this sense, $p(s, T_b)$ strives a balance between shorter traces and longer traces that can reach the target sites with *two heterogeneous metrics*. Admittedly, there may still exist some bias. One of the scenarios is that the power function may assign more energy to a seed that covers many functions in the target function trace closure. For example, assume two traces that can reach the target function T: $\langle a, b, c, d, T, Z\rangle$ and $\langle a, e, f, g, T, Z\rangle$; and the target function trace closure is $\langle a, b, c, d, e, f, g, T\rangle$. The power function may assign much energy to a seed with trace $\langle a, b, c, d, e, f, g, Z\rangle$ which does not reach the target function T. This is *not* an issue in our opinion: since this seed has covered many "expected" functions, it has high chance to be "close" to the target; with proper mutations, it is likely to be flipped to mutants that can indeed touch the target.

In Hawkeye, the power function determines the number of mutation chances to be applied on the current seed (i.e., energy); it is also used during the seed prioritization to determine whether the *mutated* seeds should be favored.

4.5 Adaptive Mutation

In §4.4, for each seed, the output of power scheduling is the energy (a.k.a. the times of applied mutations), which will be the input of the step of our adaptive mutation. The problem is that, given the total energy available for a seed, we still need to assign the number of mutations for *each type of mutators*.

In general, two categories of mutators are used in GFs. Some are coarse-grained in the sense that they change bulks of bytes during the mutations. Others are fine-grained since they only involve a few byte-level modifications, insertions or deletions. For coarse-grained mutations, we consider them to be:

(1) **Mixed havoc**. This includes several bulk mutations, namely deleting a chunk of bytes, overwriting the given chunk with other bytes in the buffer, deleting a certain lines, duplicating certain lines multiple times, etc. The actual mutation involves their combinations.
(2) **Semantic mutation**. This is used when the target program is known to process semantic relevant input files such as javascript, xml, css, etc. In detail, this follows Skyfire [43], which includes three meta mutations, inserting another subtree into a random AST position, deleting a given AST, and replacing the given position with another AST.
(3) **Splice**. This includes a crossover between two seeds in the queue and subsequent mixed havocs.

Algorithm 1: *adaptiveMutate*(): Adaptive Mutation

```
input  : s, the seed to be fuzzed after power scheduling
output : M_s, the map to store the new mutated seed, whose key is
         the seed and whole value is the energy of the seed
const. : γ, the constant ratio to do fine-grained mutation
const. : δ, the constant ratio to be adjusted
1  M_s = ∅;
2  p ← s.getScore();
3  if reachTarget(s) == false then
4      S' ← coarseMutate(s, p * (1 − γ));
5      for s' in S' do
6          M_s ← M_s ∪ {(s', s'.getScore())}
7      S'' ← fineMutate(s, p * γ);
8      for s'' in S'' do
9          M_s ← M_s ∪ {(s'', s''.getScore())}
10 else
11     S' ← coarseMutate(s, p * (1 − γ − δ));
12     for s' in S' do
13         M_s ← M_s ∪ {(s', s'.getScore())}
14     S'' ← fineMutate(s, p * (γ + δ));
15     for s'' in S'' do
16         M_s ← M_s ∪ {(s'', s''.getScore())}
```

Algo. 1 shows the workflow of our adaptive mutation, given a seed s. The basic idea is to give less chance of coarse-grained mutations when the seed s can reach the target functions (at line 10 in Algo. 1). Once the seed reaches targets, the times of doing fine-grained mutations increase from $p * \gamma$ (line 7) to $p * (\gamma + \delta)$ (line 14), but the times of doing coarse-grained mutation decrease from $p * (1 - \gamma)$ (at line 4) to $p * (1 - \gamma - \delta)$ (line 11). Here, *s.getScore*() at line 2 is to get the energy assigned to the seed according to the the power function value calculated in Equation 6.

fineMutate() in Algo. 1 simply applies a random fine-grained mutation (e.g., bit/byte flippings, arithmetics on some bytes) for the seed. Algo. 2 shows the details for coarse-grained mutation strategy *coarseMutate*(). Given a seed s and the iteration times of mutations i, the basic idea is to apply semantic mutations (line 2) only when it is necessary (line 3). The constraints *needSemMutation*(s) returns true if the following conditions are satisfied: 1) our fuzzer detects that the input file is a semantic-relevant input file such as javascript, xml, css, etc; 2) The previous semantic mutations have not failed. If not necessary (line 6), mixed havoc mutations will get more times

Algorithm 2: *coarseMutate*(): Coarse-Grained Mutation

input : s, the seed to be fuzzed after power scheduling
input : i, the number of iterations to do mutation on the seed
output: $\mathcal{S}$, the set to store the *new* mutated seed
const. : σ, the constant ratio to do semantic mutations
const. : ζ, the constant ratio to do mixed havoc mutations

1 $\mathcal{S} = \emptyset$;
2 **if** $needSemMutation(s) == true$ **then**
3 $\quad \mathcal{S} \leftarrow \mathcal{S} \cup semMutate(s, i * \sigma)$;
4 $\quad \mathcal{S} \leftarrow \mathcal{S} \cup coarseHavoc(s, i * (1-\sigma) * \zeta)$;
5 $\quad \mathcal{S} \leftarrow \mathcal{S} \cup splice(s, i * (1-\sigma) * (1-\zeta))$;
6 **else**
7 $\quad \mathcal{S} \leftarrow \mathcal{S} \cup coarseHavoc(s, i * \zeta)$;
8 $\quad \mathcal{S} \leftarrow \mathcal{S} \cup splice(s, i * (1-\zeta))$;

Algorithm 3: *seedPrioritize*(): Seed Prioritization

input : s, the seed to be processed
output: Q_1, the tier 1 queue to store the most important seeds
output: Q_2, the tier 2 queue to store the important seeds
output: Q_3, the tier 3 queue to store the least important seeds
const. : η, the threshold of energy value for accepting important seeds

1 $Q_1 = Q_2 = Q_3 = \emptyset$;
2 **if** $seedIsNew(s) == true$ **then**
3 $\quad$ **if** $seedWithNewEdge(s) == true$ **then**
4 $\quad\quad Q_1 \leftarrow Q_1 \cup \{s\}$;
5 $\quad$ **else if** $s.powerEnergy() > \eta$ **then**
6 $\quad\quad Q_1 \leftarrow Q_1 \cup \{s\}$;
7 $\quad$ **else if** $reachTarget(s) == true$ **then**
8 $\quad\quad Q_1 \leftarrow Q_1 \cup \{s\}$;
9 $\quad$ **else**
10 $\quad\quad Q_2 \leftarrow Q_2 \cup \{s\}$;
11 **else**
12 $\quad Q_3 \leftarrow Q_3 \cup \{s\}$;

($i * \zeta$, at line 7) than the necessary case ($i * (1-\sigma) * \zeta$, at line 4), and meanwhile splice mutations will also get more times ($i * (1-\zeta)$ line 8) than necessary ($i * (1-\sigma) * (1-\zeta)$, at line 5).

In practice, we assign the empirical values to the constants: $\gamma = 0.1$, $\delta = 0.4$, $\sigma = 0.2$, $\zeta = 0.8$.

Note that all these new generated seeds in $\mathcal{M}_s$, together with the original seeds, will be put into the seed queue for future fuzzing. Actually, before fuzzing them, we will prioritize them to improve the efficiency of directed fuzzing (see §4.6).

4.6 Seed Prioritization

Not all the seeds have equal or similar priorities, ideally the queue that stores the seeds to be mutated should be a priority queue. However the scoring may be biased (due to the limitations of static analyses, etc.), and the insertion operations on priority queue take a complexity of $\Theta(\log n)$, which is costly since the queue can be quite long and the insertion operation can be frequent. Therefore it is not beneficial *in practice*.[3] Instead, we provide a three-tiered queue which appends newly generated seeds into different categories according to their scores. Seeds in the top-tiered queue (tier 1) will be picked firstly, then the second-tiered (tier 2), and finally the lower-tiered (tier 3). This imitates a simplified priority queue with constant time complexity.

Algo. 3 shows the seed prioritization strategy for a new seed mutated from the previous step of adaptive mutation. The basic idea is: we should prioritize the *newly generated* seeds that 1) cover new traces 2) have bigger similarity values with the target seeds (i.e., power function values) 3) cover the target functions. We favor the seeds that cover the new traces since we still have to explore *more execution paths* that have the potential to lead to the target sites; this is necessary when the initial seeds are quite far from the targets. The other two prioritization strategies, i.e., comparing similarity values and checking whether the target function have been reached, are specific to directed fuzzing. Note that although these two strategies are relevant, neither of them can be deduced from the other. For the other newly generated seeds, they are put in the second-tiered queue. On the other hand, seeds that have (just) been mutated are assigned with the least priority. In practice, Hawkeye also applies AFL's loop bucket approach (see [49]) to filter out a large number of "equivalent" seeds that do not bring new coverage in terms of loop iterations. The prioritization strategies will be applied on the remaining seeds. Therefore, there will not be too many seeds filling up the top-tier queue.

By combining all the static and dynamic techniques mentioned above, for the CVE exampled in §2.1, Hawkeye successfully reproduced the crash with a time budget of 24 hours in 3 out of the 10 runs we conducted when fed with the same initial seeds, which is a significantly improvement on both AFL and AFLGo for this case.

5 EVALUATION

We implemented our static instrumentation on top of AFL's LLVM mode and the pointer analysis is based on the interprocedural static value-flow analysis tool called SVF [41]; this part takes about 2000 lines of C/C++ code. The dynamic fuzzer is implemented based on our Rust implementation of AFL. The fundamental framework, called Fuzzing Orchestration Toolkit[11], is written in about 14000 lines of code. We follow exactly AFL's practice [49] by using fork-server, shared memory based basic block transition (edge) tracing, deterministic/non-deterministic mutators, etc., and provide a similar user interface as AFL's. The differences, however, are that we design the fuzzer with considerations of modularization and extensibility without sacrifying performance. For directed fuzzing purpose, we add another 4000 lines of code for tracing functions, calculating power function for seeds, distinguishing graininess of mutators, and so on [4]. See our website https://sites.google.com/view/ccs2018-fuzz for more details.

For each program with the given target sites, the instrumentation of Hawkeye consists of three parts: 1) basic block IDs that track the execution traces 2) basic block distance information that determines basic block trace distance and 3) function IDs that track functions that have been covered.

5.1 Evalution Setup

In the experiments, we aim to answer the following questions:

[3] In fact, the well-known AFL fuzzer only maintains a linked list with some probabilistic to skip seeds that do not cover new edges; the complexity of the insertion is $\Theta(1)$.

[4] The semantic mutation part is implemented with Antlr [31] in Java (~4800 lines of code) that is called from Rust via JNI. We didn't enable this mutation strategy in the experiments since this is not tightly relevant to DGF.

Table 1: Program statistics for our tested programs.

Project	Program	Size	ics	cs	ics/cs	# of C_B>1	# of C_N>1	t_s
Binutils	cxxfilt	2.8M	3232	12117	26.67%	8813	8879	735s
Oniguruma	testcu	1.3M	556	2065	26.93%	3037	3101	5s
mjs	mjs	277K	130	3277	3.97%	309	334	3s
libjpeg	libjpeg	810K	749	1827	41.00%	144	152	2s
libpng	libpng	228K	449	1018	44.11%	61	61	2s
freetype2	freetype	1.6M	627	5681	11.30%	6784	7117	4s

RQ1 Is the static analysis really worth the effort?
RQ2 How good is Hawkeye's performance in terms of reproducing the target crashes?
RQ3 How effective are the dynamic strategies in Hawkeye?
RQ4 How good is the ability of Hawkeye for reaching the specific target sites?

Evaluation Dataset. We evaluated Hawkeye with diverse real-world programs:

(1) **GNU Binutils** [5] is a collection of binary analysis tools used in GNU/Linux platform. This benchmark is also used in several other works such as [6, 7, 24].
(2) **MJS** [39] is an embedded JavaScript engine for C/C++ and used in IoT development. It is used to compare Hawkeye directly with AFLGo due to implementation limitations of the latter.
(3) **Oniguruma** [23] is a versatile regular expression library used by multiple world famous projects such as PHP [33].
(4) **Fuzzer Test Suite** [18] is a set of benchmarks for fuzzing engines. It contains several representative real-world projects.

Evalution Tools. We compare Hawkeye with the following three fuzzers:

(1) **AFL** is the current state-of-the-art GF. It ignores all the target information for the PUT and only does the "basic block transition" instrumentation.
(2) **AFLGo** is the state-of-the-art DGF based on AFL. Compared to AFL, it also instruments basic block distance information.
(3) **HE-Go** is the fuzzer where the basic block level distance is generated with our static analysis procedure (Fig. 3), but the dynamic fuzzing is conducted by AFLGo.

Here we mainly follow AFLGo's practice to only use AFL as the baseline for coverage oriented GFs. Other techniques do not focus on directed fuzzing, and they are either orthogonal (e.g., CollAFL [15]) or may sometimes perform worse than AFL (e.g., AFLFast, as observed by [38]), or not publicly available (e.g., Angora [12]). The detailed reason is available at our website; in §6, we also provide a more detailed comparison between Hawkeye and these techniques.

In the experiments, all AFL based fuzzers (AFL, AFLGo and HE-Go) are run in their "fidgety" mode [50]. For both AFLGo and HE-Go, "time-to-exploitation" is set to 45 minutes for the fuzzer. Except for the experiments against GNU Binutils (Table 2) , where we follow exactly the setup in AFLGo's paper [6] , all the other experiments are repeated 8 times, with a time budget of 4 hours. We use "time-to-exposure" (TTE) to measure the length of the fuzzing campaign until the first test input is generated that triggers a given error (in §5.3) or reaches a target site (in §5.4). We use *hitting round* to measure the number of runs in which a fuzzer triggers the error or reaches the target. For all the experiments, if the fuzzer cannot find the target crash within the time budget in one run, TTE is set to the time budget value.

Our experiments are conducted on an Intel(R) Xeon(R) CPU E5-2697 v3 @ 2.60GHz with 28 cores, running a 64-bit Ubuntu 16.04 LTS system; during experiments, we use 24 cores and retain 4 cores for other processes.

5.2 Static Analysis Statistics

In Table 1, the first three columns denote the projects, programs and the sizes in their LLVM bitcode form. *ics* denotes the number of indirect call sites in the binary, which is calculated by counting those call sites without explicitly known callees; *cs* is the number of call sites; *ics/cs* denotes the percentage of indirect calls among all call sites. The next two columns denote the number where $C_B > 1$ and $C_N > 1$ (§ 4.2), respectively. The last column denotes the time cost of call graph generation, which takes the majority of the time among all the directedness utility computation.

We can see from the table that the chosen targets have fair diversities in terms of different metrics. It is also noticeable that the number of indirect function calls may contribute a large portion to the total number of function calls. Specifically, in libpng, 44.11% function calls are indirect function calls. This clearly shows the importance of building precise call graphs. Furthermore, the number of occurrences of $C_N > 1$ and the number of occurrences of $C_B > 1$ are also large, which shows the importance of taking into consideration the different patterns of call relations.

As to the overhead of the static directedness utility computation, except for *cxxfilt*, which requires approximately 12.5 minutes to generates the call graph, call graphs of most other projects can be generated in seconds. For cxxfilt, the performance degradation lies in the inherent complexity of the project itself. From the program statistics in Table 1), it is obvious that the code base is bigger and the program structures are more complicated than the others. In fact, the bottleneck of the analysis is inside the pointer analysis implemented in SVF tool. We believe that it is worth the effort due to the fact that this procedure is done purely statically. And as long as the source code does not change, the call graph can be reused.

5.3 Crash Exposure Capability

The most common application of directed fuzzing is to try to expose the crash with some given suspicious locations that are supposed to be vulnerable, where the suspicious locations can be detected with the help of other static or dynamic vulnerability detection tools. In this experiment, we directly compare Hawkeye with other fuzzers on some known crashes to evaluate its crash exposure capability.

Table 2: Crash reproduction in Hawkeye, AFLGo and AFL against Binutils.

CVE-ID	Tool	Runs	μTTE(s)	Factor
2016-4487 2016-4488	Hawkeye	20	177	–
	AFLGo	20	390	2.20
	AFL	20	630	3.56
2016-4489	Hawkeye	20	206	–
	AFLGo	20	180	0.87
	AFL	20	420	2.04
2016-4490	Hawkeye	20	103	–
	AFLGo	20	93	0.90
	AFL	20	59	0.57
2016-4491	Hawkeye	9	18733	–
	AFLGo	5	23880	1.27
	AFL	7	20760	1.11
2016-4492 2016-4493	Hawkeye	20	477	–
	AFLGo	20	540	1.21
	AFL	20	960	2.01
2016-6131	Hawkeye	9	17314	–
	AFLGo	6	21180	1.22
	AFL	2	26340	1.52

5.3.1 Crash Reproduction against Binutils. In the beginning, we intended to compare GNU Binutils directly with AFLGo in our experiments since it is an important benchmark in [6] to demonstrate AFLGo's directedness. However, we found that the actual implementation of AFLGo has a few issues [2] in generating static distances. Most importantly, it takes too long to calculate the distances. As a result, when we tried AFLGo's static analysis on GNU Binutils 2.26, it failed to generate "distance.cfg.txt" which contains the distance information for instrumentation within 12 hours[5]. Although AFLGo can still perform fuzzing without distance information instrumentation, the fuzzing process is no longer *directed* without any distance input. Therefore, we reclaimed the results in [6] to compare with ours for the GNU Binutils benchmark [6]. We follow exactly the evaluation setup in [6] where each experiment is conducted for 20 times, with the time budget set as 8 hours; the initial input seed file only contains a line break (generated by `echo "" > in/file`). The target sites we specified are based on their CVE descriptions and the backtraces of the crashes. We compare Hawkeye with AFLGo and AFL; the results are shown in Table 2. In AFLGo's paper, the A_{12} metric [42] is used to show the possibility that one fuzzer is better than the other according to all the runs. It is ignored in Table 2 since we cannot get the result of each run in their experiments.

We can observe the following facts: 1) For CVE-2016-4491 and CVE-2016-6131, Hawkeye achieves the best results, with the most hitting rounds (both are 9 rounds) and the shortest μ**TTE** (18773s and 17314s). Compared with other tools, on average for both cases, Hawkeye's improvements are significant in term of hitting rounds (> 20%) and μ**TTE** (> 20%). 2) For CVE-2016-4487/4488 and CVE-2016-4492/4493, all tools reproduce the crashes in 20 runs, and Hawkeye achieves the best μ**TTE**. Specifically, on these two cases, Hawkeye's improvement in terms of μ**TTE** is significant — reducing at least 20% than other tools. 3) For CVE-2016-4489 and CVE-2016-4490, all tools reproduce the crashes in all runs within 7 minutes since these bugs are relatively easy to find. Apparently, in such cases, directed fuzzers have no significant advantage — in other words, when the crashes are shallow or easy to trigger, Hawkeye's merits cannot show and fuzzing randomness matters for μ**TTE**.

[5]Besides the performance issue of AFLGo, another reason to reclaim AFLGo's results in Table 2 is that the hardware environments are similar. The reason is supported by the similar results produced by AFL in **[6]** and our experiments.

[6]Since CVE-2016-4487/CVE-2016-4488 and CVE-2016-4492/CVE-2016-4493 share the same target sites, we treat them as the same; the reclaimed value for CVE-2016-4487/CVE-2016-4488 are also average values.

Table 3: Crash reproduction in Hawkeye, AFLGo and AFL against MJS.

Bug ID	Tool	Runs	μTTE(s)	Factor	A_{12}
#1	Hawkeye	5	5469	–	–
	AFLGo	2	12581	2.30	0.77
	AFL	2	13084	2.39	0.77
#2	Hawkeye	7	1880	–	–
	AFLGo	2	12753	6.78	0.95
	AFL	2	12294	6.54	0.95
#3	Hawkeye	8	178	–	–
	AFLGo	8	819	4.60	0.91
	AFL	8	1269	7.13	0.95
#4	Hawkeye	8	5519	–	–
	AFLGo	8	5878	1.07	0.57
	AFL	8	5036	0.91	0.48

To summarize, Hawkeye has the real potential to fulfill directed fuzzing tasks where the target crashes are not easy to be detected.

5.3.2 Crash Reproduction on MJS. In order to *directly* compare the performance between Hawkeye and AFLGo, we chose a project called *MJS*, which contains a single source file and the results are in Table 3. We used this project for direct comparison with AFLGo since AFLGo took too much time or failed to generate the distance information for other projects such as Oniguruma, libpng, etc. On *MJS*, AFLGo took an average of 13 minutes to generate the basic block distance for different targets. During experiments, the initial input seed files are all from the project's *tests* directory. The targets are selected from the crashes reported in the project's GitHub pages, which correspond to four categories of vulnerabilities, namely *integer overflow* (#1), *invalid read* (#2), *heap buffer overflow* (#3), and *use after free* (#4). We can observe the following facts : 1) On #1 and #2, Hawkeye achieves the best results, with the most hitting rounds and the shortest μ**TTE** for both cases. In terms of hitting rounds, Hawkeye found #1's bug in 5 runs and #2's bug in 7 runs, while for the other two tools they only detected both crashes in 2 runs. Notably, this case is nontrivial and Hawkeye reduces the μ**TTE** from about 3.5 hours to 0.5 hours. 2) On #3, for which all the tools reproduce the crash in 8 rounds. Still, Hawkeye has the highly significant improvement on μ**TTE**, using less than one fourth μ**TTE** of other tools. 3) On #4, all the tools reproduce the crash in all rounds, and the μ**TTE** differences among them are not significant. As to A_{12}, we can see that Hawkeye exhibits really good results, for example, the values in #2 are both 0.95, which means Hawkeye has 95% confidence to perform better than both other tools.

5.3.3 Crash Reproduction on Oniguruma. Here we compare Hawkeye with HE-Go on Oniguruma to show the advantage of dynamic analysis strategies in Hawkeye. Therefore, the major differences between Hawkeye and HE-Go is mainly the dynamic part. Due to the aforementioned performance issues of AFLGo in static analysis on

Table 4: Crash reproduction on Hawkeye, HE-Go and AFL against Oniguruma.

Bug ID	Tool	Runs	μTTE(s)	Factor	A_{12}
#1	Hawkeye	8	139	–	–
	HE-Go	8	149	1.07	0.58
	AFL	8	135	0.97	0.54
#2	Hawkeye	8	186	–	–
	HE-Go	8	228	1.23	0.88
	AFL	8	372	2.00	1.0
#3	Hawkeye	2	13768	–	–
	HE-Go	1	14163	1.03	0.56
	AFL	1	14341	1.04	0.57
#4	Hawkeye	7	6969	–	–
	HE-Go	3	12547	1.80	0.82
	AFL	1	14375	2.06	0.88

Oniguruma and other big projects, hereinafter, we will used HE-Go as an alternative to AFLGo in the subsequent experiments. We therefore compare Hawkeye with HE-Go to show the effectiveness of our dynamic strategies.

In Table 4, we compare Hawkeye with HE-Go and AFL against the Oniguruma regex library. The first three bugs come from the reported CVEs which occur on version 6.2.0, and Bug #4 is an newly fixed vulnerability issue on its GitHub pages. Some observations are: 1) For #3 and #4, Hawkeye achieves the best results among the three tools. Especially, for #4, the improvements in both hitting rounds and μ**TTE** are highly significant. For #3, Hawkeye can find the bug in one more round, but the μ**TTE** is similar for the three tools. 2) For #1 and #2, all the tools reproduce the crash in 8 rounds. On the other hand, Hawkeye and HE-Go have no significant differences in μ**TTE**. From the results, we can conclude that the dynamic analysis strategies used in Hawkeye are effective.

5.4 Target Site Covering

Certain locations in the PUTs are hard to reach, though they may not trigger crashes. In practice, the generated seeds that can cover the such locations can also be used as initial seeds for CGFs to boost their coverage. Therefore, this criterion is an important factor for measuring DGFs' capabilities.

Google's fuzzing test suite contains three projects that specially focus on testing fuzzers' abilities of discovering hard to reach locations, namely *libjpeg-turbo-07-2017* (#1), *libpng 1.2.56* (#2, #3) and *freetype2-2017* (#4). In these benchmarks, the target sites are specified by file names with line numbers in the source files. Here we manually added some additional "sentinel" code in the target sites ("exit(n)", where the values of 'n' distinguish these sites) to indicate that the relevant targets have been reached.

Table 5 shows the results on these benchmarks. Case #1 and #4 show that Hawkeye exhibits good capability in terms of rapidly covering the target sites, according to μ**TTE** and the factor columns; considering A_{12}, the behaviors are also steady. In case #2 and #3, it takes little time to reach these target sites for all the fuzzers. While on the relevant project page [14], it is mentioned clearly that they "currently require too much time to find". We actually tried this benchmark on libFuzzer with the default scripts in 2 machines, indeed it failed to reach the target sites. This root cause of the inconsistency may lie in the fact that the inner mechanisms

Table 5: Target site covering results in Hawkeye, HE-Go and AFL against Fuzzer Test Suite (libjpeg-turbo, libpng, freetype2).

ID	Project	Tool	Runs	μTTE(s)	Factor	A_{12}
#1	jdmarker.c:659	Hawkeye	8	1955	–	–
		HE-Go	8	2012	1.03	0.53
		AFL	8	4839	2.48	0.95
#2	pngread.c:738	Hawkeye	8	23	–	–
		HE-Go	8	16	0.70	0.43
		AFL	8	130	5.65	1.00
#3	pngrutil.c:3182	Hawkeye	8	1	–	–
		HE-Go	8	66	66.00	0.56
		AFL	8	3	3.00	0.51
#4	ttgload.c:1710	Hawkeye	7	4283	–	–
		HE-Go	7	4443	1.04	0.55
		AFL	6	5980	1.40	0.60

may affect the actual fuzzing effectiveness (In fact, libFuzzer is known to be quite different from AFL and its derivations). The other observation is that HE-Go has a rather big value in terms of μ**TTE** compared to other tools. It turns out that in one of the runs, the TTE is 524s, much larger than all the other runs.

It is worth noting that the μ**TTE** to cover the target sites (Table 5) is *quite* different from the μ**TTE** to trigger real-world crashes (Table 4): the TTEs of the former are calculated based on the duration to cover the specific *line* at the first time; while the TTEs of the latter are tightly relevant to *branch coverage* or even *path coverage* since typically bugs in widely used software can only be triggered with special path conditions and it requires covering a few execution traces. Although Table 5 shows that Hawkeye's improvements against HE-Go in covering target sites are not obvious (and for a few cases, it performs worse), we can observe in Table 4 the acceleration on crash reproduction is significant specially for #4 (1.80x, nearly 2 hours off) and #2 (1.23x). This actually indicates that dynamic strategies are quite effective in *detecting crashes*.

5.5 Answers to Research Questions

With the experiments conducted in Tables 2, 3, 4 and 5, we are able to answer the research questions.

RQ1 We consider it is worth to apply static analysis. As shown in Table 1, the time cost of our static analysis is generally acceptable compared to the runtime cost during fuzzing. Even for the cxxfilt cases in Table 2, which takes on average 735 seconds, Hawkeye outperforms the vanilla AFL in most of the cases. Two notable results are CVE-2016-4491 and CVE-2016-6131, it saves roughly 2000s and 9000s to detect the crash; as shown from the A_{12} metric, the results are also consistent in all the 20 runs. On the other hand, Hawkeye also demonstrates some boosts for fuzzing.

RQ2 Hawkeye performs quite well in detecting crashes. From the results in Tables 2, 3 and 4, we can clearly see that Hawkeye can detect the crashes more quickly than all the other tools; the results are even steady among different runs as shown by different A_{12} results.

RQ3 The dynamic strategies used in Hawkeye are quite effective. It is obvious that in all the experiments we conducted, Hawkeye

outperforms the others. In particular, the experiments in comparison with HE-Go (Table 4 and 5) show that our combination of power scheduling, adaptive mutation strategies, and seed prioritization make Hawkeye converge faster than AFLGo's simulated annealing based scheduling.

RQ4 From the results in Table 5, we are confident that Hawkeye has the capability to reach the target sites rapidly.

In practice, Hawkeye also demonstrates its power in exposing crashes with the help of other vulnerability detection tools. For example, for Oniguruma and MJS projects, with the Clang Static Analyzer [27] (the built-in and our customized checkers) reporting suspicious vulnerability locations (i.e., target sites) in the programs, Hawkeye successfully detected the crashes by directing the fuzzing to those locations. Interestingly, for MJS, we marked several of the authors' newly patched program locations, and detected a few other crashes even further. As a result, Hawkeye has reported more than 28 crashes in projects Oniguruma and MJS. We have also found multiple vulnerabilities in other projects such as Intel XED x86 encoder decoder (4 crashes), Espruino JavaScript interpreter (9 crashes). All these crashes have been confirmed and fixed, and 15 of them have been assigned with CVE IDs.

5.6 Threats to Validity

The internal threats of validity are twofold: 1). Several components of Hawkeye (e.g., Algo. 1 and 2) utilize the predefined thresholds to make decision. Currently, these thresholds (e.g., $\gamma = 0.1$, $\delta = 0.4$, $\sigma = 0.2$, $\zeta = 0.8$) are configured according to our preliminary experiments and previous experience in fuzzing. Systematic research will be planned to investigate the impact of these thresholds and figure out the best configurations. 2). As we rely on the lightweight program analysis tools like LLVM and SVF [41] to calculate the distance, possible issues of these tools may affect the final results. As Hawkeye is well modularized and can easily integrate with other static analysis tools, enhancing Hawkeye with other tools will be another alternative solution.

The external threats rise from the choice of evaluation dataset and the CVEs for crash reproduction. Despite we adopt the program Binutils that is used in AFLGo [6], the evaluation results still need to be generalized with an empirical study on more projects in future. Besides, the tested CVEs in *MJS* and *Oniguruma* are *not selectively* chosen for the purpose to show the advance of Hawkeye— we pick them since they are reported within a recent period.

6 RELATED WORK

Our study is related to the following lines of research:

Directed Grey-box Fuzzing. Some other DGF techniques have been proposed besides Hawkeye. AFLGo [6] is the state-of-the-art directed grey-box fuzzer which utilizes a simulated annealing-based power schedule that gradually assigns more energy to inputs that hold the trace closer to the target sites. In AFLGo, the authors proposed a novel idea of calculating the distance between the input traces and the target sites. This is a good starting point by combining such target distance calculation with grey-box fuzzer. Hawkeye is inspired from AFLGo however provides significant improvements on both the static analysis and dynamic fuzzing. As shown in §5, Hawkeye generally outperforms AFLGo in terms of reaching the targets and reproducing crashes, thanks to embedding in-depth consideration about the four desired properties into the design. SeededFuzz [45] uses various program analysis techniques to facilitate the generation and selection of initial seeds which helps to achieve the goal of directed fuzzing. Equipped with the improved seed selection and generation techniques, SeededFuzz can reach more critical sites and find more vulnerabilities. The core techniques of SeededFuzz are orthogonal to Hawkeye because SeededFuzz focuses on the quality of initial seed inputs while Hawkeye focuses on the four desirable properties regardless of initial seeds.

Note that our proposed four properties can also be applied for DGF on programs where the source code is unavailable. In fact, we are extending Hawkeye to be able to work on the binary-only fuzzing scenarios. Technically, the target sites can be determined by binary-code matching techniques on attack surface identification [9, 47]; the static analysis can be achieved with binary analysis tools such as IDA [20]; and the instrumentation can be done by dynamic binary instrumentators such as Intel Pin [1]. We envision the extended Hawkeye can piggyback on these techniques and demonstrate its effectiveness even further.

Directed Symbolic Execution. Directed Symbolic Execution (DSE) is one of the most related techniques to DGF as it also aims to execute target sites of the PUT. Several works have been proposed for DSE [17, 19, 21, 28, 29]. These DSE techniques rely on heavyweight program analysis and constraint-solving to reach the target sites systematically. A typical example of DSE is Katch [29], which relies on symbolic execution, augmented by several synergistic heuristics based on static and dynamic program analysis. Katch can effectively find bugs in incomplete patches and increase the patch coverage comparing to manual test suite. However, as discussed in [6], DGF is generally more effective on real-world programs as DSE techniques suffer from the infamous path-explosion problem [40]. In contrast to DSE, Hawkeye relies on lightweight program analysis, which ensures its scalability and execution efficiency.

Taint Analysis Aided Fuzzing. Taint analysis is also widely used to facilitate directed white-box testing [12, 16, 24, 34, 44]. The key intuition of using taint analysis in fuzzing is to identify *certain parts* of the input which should be mutated with priority. In such a way, the fuzzer can drastically reduce the search space for reaching certain desired locations. Taint based approaches are more scalable than the DSE techniques and can help the fuzzer to reach certain preferable locations such as rare branches in Fairfuzz [24] or checksum related code in TaintScope [44]. Different from Hawkeye, these techniques are not fed with *given target sites* (e.g., file name and line numbers in our scenario) but based on source-sink pairs. Thus, such techniques do not have advantages in scenarios where the targets are clear, such as patch testing and crash reproduction.

Coverage-based Grey-box Fuzzing. The purposes of coverage-based grey-box fuzzing (CGF) and DGF are different. However, some techniques proposed to boost the performance of CGF could also be adopted by Hawkeye. For example, CollAFL [15] utilizes a novel hash algorithm to solve AFL's instrumentation collision problem. Skyfire [43] learns a probabilistic context sensitive grammar (PGSG) to specify both syntax features and semantic rules, and then the second step leverages the learned PCSG to generate new test seeds. Xu *et al.* [46] proposed a set of new operating primitives to improve the performance of grey-box fuzzers. Another important topic in CGF is about guiding the fuzzer through path constraints. [12, 25,

32, 34, 40] aim to help the CGFs to break through path constraints. Moreover, Orthrus [38] applies static analysis on AST, CFG, and CG to extract complicated tokens via customizable queries. Hawkeye can benefit through combining with the aforementioned techniques.

7 CONCLUSIONS

In this paper, we propose a novel directed grey-box fuzzer, Hawkeye. The design of Hawkeye embeds four desired properties for directed fuzzing by combining static analysis and dynamic fuzzing in an effective way. Equipped with a better evaluation of the distance between input execution traces and the user specified target sites, Hawkeye can precisely and adaptively adjust its seed prioritization, power scheduling as well as mutation strategies to reach the target sites rapidly. A thorough evaluation showed that Hawkeye can reach the target sites and reproduce the crashes much faster than existing state-of-the-art grey-box fuzzers. The promising results indicate that Hawkeye can be effective in patch testing, crash exposure and other scenarios.

ACKNOWLEDGMENT

This work is supported by the National Research Foundation, Prime Ministers Office, Singapore under its National Cybersecurity R&D Program (Award No. NRF2016NCR-NCR002-026) and administered by the National Cybersecurity R&D Directorate; the research of Dr Xue is also supported by CAS Pioneer Hundred Talents Program.

REFERENCES

[1] 2018. Pin - A Dynamic Binary Instrumentation Tool. https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool
[2] AFLGo. 2018. GitHub - AFLGo. https://github.com/aflgo/aflgo/issues
[3] Lars Ole Andersen. 1994. *Program Analysis and Specialization for the C Programming Language.* Technical Report. DIKU, University of Copenhagen.
[4] Jean-Yves Audibert, Rémi Munos, and Csaba Szepesvári. 2009. Exploration-exploitation tradeoff using variance estimates in multi-armed bandits. *Theoretical Computer Science* 410, 19 (2009), 1876 – 1902. http://www.sciencedirect.com/science/article/pii/S030439750900067X Algorithmic Learning Theory.
[5] GNU Binutils. 1990. GNU Binutils. https://www.gnu.org/software/binutils/
[6] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed Greybox Fuzzing *(CCS '17)*. ACM Press, New York, NY, USA, 2329–2344.
[7] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2016. Coverage-based Greybox Fuzzing As Markov Chain *(CCS '16)*. ACM Press, New York, NY, USA, 1032–1043.
[8] Denny Britz. 2014. Exploitation vs Exploration. https://medium.com/@dennybritz/exploration-vs-exploitation-f46af4cf62fe
[9] Mahinthan Chandramohan, Yinxing Xue, Zhengzi Xu, Yang Liu, Chia Yuan Cho, and Hee Beng Kuan Tan. 2016. BinGo: Cross-architecture cross-OS Binary Search *(FSE '16)*. ACM Press, New York, NY, USA, 678–689.
[10] Chen Chen, Baojiang Cui, Jinxin Ma, Runpu Wu, Jianchao Guo, and Wenqian Liu. 2018. A systematic review of fuzzing techniques. *Computers & Security* 75 (2018), 118–137.
[11] Hongxu Chen, Yuekang Li, Bihuan Chen, Yinxing Xue, and Yang Liu. 2018. FOT: A Versatile, Configurable, Extensible Fuzzing Framework *(FSE '18 tool demo)*. ACM Press, (to appear).
[12] Peng Chen and Hao Chen. 2018. Angora: Efficient Fuzzing by Principled Search. *CoRR* abs/1803.01307 (2018). arXiv:1803.01307 https://arxiv.org/abs/1803.01307v2
[13] J. B. Crawford. 2018. A survey of some free fuzzing tools. https://lwn.net/Articles/744269/
[14] fuzzer-test suite. 2018. libpng-1.2.56/test-libfuzzer.sh. https://github.com/google/fuzzer-test-suite/blob/master/libpng-1.2.56/test-libfuzzer.sh
[15] Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuwen Tu, Kang Li, Zhongyu Pei, and Zuoning Chen. 2018. CollAFL: Path Sensitive Fuzzing *(SP '18)*. IEEE Press, 1–12.
[16] Vijay Ganesh, Tim Leek, and Martin Rinard. 2009. Taint-based Directed Whitebox Fuzzing *(ICSE '09)*. IEEE Computer Society, Washington, DC, USA, 474–484.
[17] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: Directed Automated Random Testing *(PLDI '05)*. ACM Press, New York, NY, USA, 213–223.
[18] Google. 2017. Fuzzer Test Suite. https://github.com/google/fuzzer-test-suite
[19] Istvan Haller, Asia Slowinska, Matthias Neugschwandtner, and Herbert Bos. 2013. Dowsing for Overflows: A Guided Fuzzer to Find Buffer Boundary Violations *(SEC '13)*. USENIX Association, Berkeley, CA, USA, 49–64.
[20] Hex-Rays. 2013. IDA. https://www.hex-rays.com/index.shtml
[21] Wei Jin and Alessandro Orso. 2012. BugRedux: Reproducing Field Failures for In-house Debugging *(ICSE '12)*. IEEE Press, Piscataway, NJ, USA, 474–484.
[22] Scott Kirkpatrick, C Daniel Gelatt, and Mario P Vecchi. 1983. Optimization by simulated annealing. *science* 220, 4598 (1983), 671–680.
[23] K. Kosako. 2002. Oniguruma. https://github.com/kkos/oniguruma
[24] C. Lemieux and K. Sen. 2017. FairFuzz: Targeting Rare Branches to Rapidly Increase Greybox Fuzz Testing Coverage. *ArXiv e-prints* (Sept. 2017). arXiv:cs.SE/1709.07101
[25] Yuekang Li, Bihuan Chen, Mahinthan Chandramohan, Shang-Wei Lin, Yang Liu, and Alwen Tiu. 2017. Steelix: Program-state Based Binary Fuzzing *(ESEC/FSE '17)*. ACM Press, New York, NY, USA, 627–637.
[26] LLVM. 2015. libFuzzer. https://llvm.org/docs/LibFuzzer.html
[27] LLVM/Clang. 2013. Clang Static Analyzer. https://clang-analyzer.llvm.org/
[28] Kin-Keung Ma, Khoo Yit Phang, Jeffrey S. Foster, and Michael Hicks. 2011. Directed Symbolic Execution *(SAS'11)*. Springer-Verlag, Berlin, Heidelberg, 95–111.
[29] Paul Dan Marinescu and Cristian Cadar. 2013. KATCH: High-coverage Testing of Software Patches *(ESEC/FSE 2013)*. ACM Press, New York, NY, USA, 235–245.
[30] Barton P. Miller, Louis Fredriksen, and Bryan So. 1990. An Empirical Study of the Reliability of UNIX Utilities. *Commun. ACM* 33, 12 (Dec. 1990), 32–44.
[31] Terence Parr. 2018. ANTLR (ANother Tool for Language Recognition). http://www.antlr.org/
[32] Hui Peng, Yan Shoshitaishvili, and Mathias Payer. 2018. T-Fuzz: Fuzzing by Program Transformation *(SP '18)*. 697–710.
[33] PHP. 1994. PHP: Hypertext Preprocessor. http://www.php.net/
[34] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. 2017. VUzzer: Application-aware Evolutionary Fuzzing *(NDSS '17)*. 1–14.
[35] Agostino Sarubbo. 2017. binutils: NULL pointer dereference in concat_filename (dwarf2.c). https://blogs.gentoo.org/ago/2017/10/03/binutils-null-pointer-dereference-in-concat_filename-dwarf2-c
[36] Agostino Sarubbo. 2017. binutils: NULL pointer dereference in concat_filename (dwarf2.c) (INCOMPLETE FIX FOR CVE-2017-15023). https://blogs.gentoo.org/ago/2017/10/24/binutils-null-pointer-dereference-in-concat_filename-dwarf2-c-incomplete-fix-for-cve-2017-15023
[37] Konstantin Serebryany and Marcel Böhme. 2017. AFLGo: Directing AFL to reach specific target locations. https://groups.google.com/forum/#!topic/afl-users/qcqFMJa2yn4
[38] Bhargava Shastry, Markus Leutner, Tobias Fiebig, Kashyap Thimmaraju, Fabian Yamaguchi, Konrad Rieck, Stefan Schmid, Jean-Pierre Seifert, and Anja Feldmann. 2017. Static Program Analysis as a Fuzzing Aid. In *Research in Attacks, Intrusions, and Defenses*, Marc Dacier, Michael Bailey, Michalis Polychronakis, and Manos Antonakakis (Eds.). Springer International Publishing, 26–47.
[39] Cesanta Software. 2016. mjs. https://github.com/cesanta/mjs
[40] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Krügel, and Giovanni Vigna. 2016. Driller: Augmenting Fuzzing Through Selective Symbolic Execution *(NDSS '16)*. 1–16.
[41] Yulei Sui and Jingling Xue. 2016. SVF: Interprocedural Static Value-flow Analysis in LLVM *(CC '16)*. ACM Press, New York, NY, USA, 265–266.
[42] Andras Vargha, András Vargha, and Harold D. Delaney. 2000. A critique and improvement of the CL common language effect size statistics of McGraw and Wong. *Journal of Educational and Behavioral Statistics* 25, 2 (2000), 101–132.
[43] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2017. Skyfire: Data-Driven Seed Generation for Fuzzing *(SP '17)*. 579–594.
[44] T. Wang, T. Wei, G. Gu, and W. Zou. 2010. TaintScope: A Checksum-Aware Directed Fuzzing Tool for Automatic Software Vulnerability Detection *(SP '10)*. 497–512.
[45] Weiguang Wang, Hao Sun, and Qingkai Zeng. 2016. SeededFuzz: Selecting and Generating Seeds for Directed Fuzzing. 49–56.
[46] Wen Xu, Sanidhya Kashyap, Changwoo Min, and Taesoo Kim. 2017. Designing New Operating Primitives to Improve Fuzzing Performance *(CCS '17)*. ACM Press, New York, NY, USA, 2313–2328.
[47] Yinxing Xue, Zhengzi Xu, Mahinthan Chandramohan, and Yang Liu. 2018. Accurate and Scalable Cross-Architecture Cross-OS Binary Code Search with Emulation. *IEEE Trans Software Engineering* (2018), (to appear).
[48] Michal Zalewski. 2014. American Fuzzy Lop. http://lcamtuf.coredump.cx/afl/
[49] Michal Zalewski. 2014. Technical "whitepaper" for afl-fuzz. http://lcamtuf.coredump.cx/afl/technical_details.txt
[50] Michal Zalewski. 2016. "FidgetyAFL" implemented in 2.31b. https://groups.google.com/forum/#!topic/afl-users/1PmKJC-EKZ0