Constraint-guided Directed Greybox Fuzzing

Gwangmu Lee
Seoul National University
gwangmu@snu.ac.kr

Woochul Shim
Samsung Research
woochul.shim@samsung.com

Byoungyoung Lee*
Seoul National University
byoungyoung@snu.ac.kr

Abstract

Directed greybox fuzzing is an augmented fuzzing technique intended for the targeted usages such as crash reproduction and proof-of-concept generation, which gives directedness to fuzzing by driving the seeds toward the designated program locations called target sites. However, we find that directed greybox fuzzing can still suffer from the long fuzzing time before exposing the targeted crash, because it does not consider the ordered target sites and the data conditions. This paper presents constraint-guided directed greybox fuzzing that aims to satisfy a sequence of constraints rather than merely reaching a set of target sites. Constraint-guided greybox fuzzing defines a constraint as the combination of a target site and the data conditions, and drives the seeds to satisfy the constraints in the specified order. We automatically generate the constraints with seven types of crash dumps and four types of patch changelogs, and evaluate the prototype system CAFL against the representative directed greybox fuzzing system AFLGo with 47 real-world crashes and 12 patch changelogs. The evaluation shows CAFL outperforms AFLGo by 2.88x for crash reproduction, and better performs in PoC generation as the constraints get explicit.

1 Introduction

Fuzz testing [32] is one of the most effective techniques in discovering the vulnerabilities in software programs. Fuzzing keeps running a target program with a randomly generated input in hopes that the program exhibits an erroneous runtime behavior (such as memory corruptions or triggering assertions). Most fuzzing techniques leverage the coverage-guided fuzzing technique [12, 46], where its input mutation is focused on extending the code coverage, because it allows to efficiently explore the deeper level of code to be tested.

In particular, directed greybox fuzzing (DGF) [21, 22, 44] focuses on driving the testing toward a set of specific program locations, called *target sites*, which allows to intensively fuzz such locations. Compared to the coverage-guided fuzzing, DGF is particularly useful if information on such locations are available. For instance, when developers get a crash report from third-party users, developers may need to reproduce the reported crash to pinpoint the root cause of the crash. Another example includes when attackers need to generate 1-day proof-of-concept (PoC) inputs for the outdated systems with the released patch, where changed locations suggest the potential cause of the fixed crash.

However, we find that DGF techniques take a very long time to identify the targeted crash largely due to the following two limitations. First, DGF assumes that the target sites are *independent* to each other, implying that it does not consider order dependency between multiple target sites (i.e., a certain target site should be executed before another target site). The most common examples of such cases would be the use-after-free cases, where the crash occurs only when the program reaches the free location before reaching the use location that references the freed memory object. Since DGF generally favors the shorter execution paths, the lack of order-dependent target sites can easily lead DGF to bypass them.

Second, DGF does not consider the *data conditions* required for the targeted crash and overlook the seeds that satisfy such data conditions. The most intuitive examples include the buffer overflow crashes, where the seeds accessing the memory close to the boundary would have a higher chance to cause the crash than the seeds accessing further away from it. Another example is the PoC generation based on the patch, where the changed data condition may be involved in the cause of the fixed crash. Again, since DGF is not aware of the data conditions, it is likely to falsely prioritize the seeds with the control-flow based distance, which may adversely affect seed scheduling.

In this paper, we propose constraint-guided directed greybox fuzzing (CDGF) that resolves the limitations of DGF. Rather than reaching a set of target sites, CDGF aims to satisfy **a sequence of constraints** and prioritizes the seeds that better satisfy those in order. A constraint consists of a single target site and optionally a number of *data conditions*, which

*Corresponding author

is regarded as being satisfied when the program reaches the target site and satisfies the data condition at the target site. Constraints can be specified more than one, and in such cases, the constraints must be satisfied *in the specified order*.

To measure how well a given seed satisfies the constraints, CDGF defines the seed distance based on the *distance of the constraints*, as opposed to the conventional DGF that defines the seed distance as the average distance to the target sites. The distance of the constraints indicates how well a given seed satisfies the constraints (i.e., the shorter the better). CDGF prioritizes the seeds with the shorter distances so that the mutated seeds can try the next unsatisfied constraints on the basis of the already satisfied constraints, quickly yielding the desired seed that exhibits the targeted crash.

In addition, we present the algorithmic methods to *automatically* generate the constraints from the additional information sources, namely crash dumps from memory error detectors [33, 39, 41] and changelogs from patches. With the use-after-free crash dumps, we generate the constraints to drive the seeds to the free location first before the crash location, so that the program first frees the vulnerable memory object before accessing it at the crash location. With the buffer overflow crash dumps, we generate the constraints to drive the seeds to the boundary of the vulnerable buffer to increase the chance of accessing the buffer out of bound. With the patch changelogs, we generate the constraints to transform the seeds into the buggy conditions indicated by the patch changelogs. Overall, our auto-generated constraints support seven kinds of crashes and four types of changelogs in total.

To demonstrate the effectiveness of CDGF in exposing the targeted crash, we implemented CAFL based on AFL 2.52b [46] and compared CAFL with the representative DGF system AFLGo [21] using 47 crash dumps and 12 patch changelogs in various real-world programs. The evaluation shows that CAFL outperforms AFLGO by 2.88x in reproducing 47 real-world crashes, and better performs in PoC generation as the constraints are more explicit.

The main contributions of this paper can be summarized as follows:

- We present the constraint-guided directed greybox fuzzing (CDGF), which augments the conventional DGF with the ordered target sites and the data conditions.
- We automatically generate the constraints with the given additional information sources, namely crash dumps and patch changelogs, that support seven crash types and four changelog types in total.
- We implement CAFL, the prototype fuzzing system with CDGF, and demonstrate the superior performance in exposing the targeted crash compared to the representative DGF system AFLGo under various real-world crashes.

The rest of the paper is organized as follows. In §2, we provide a brief background about the conventional DGF and its limitations by CVE examples. In §3, we present the basic idea of CDGF and demonstrate how CDGF resolves the limitations manifested by the examples. In §4, we formally define the constraint itself and the distance metric of a sequence of constraints. In §5, we provide an algorithmic method to automatically generate the constraints from the additional information sources. In §6, we describe the internal organization of CAFL, the prototype fuzzing system equipped with CDGF. In §7, we compare the performance of CAFL against the representative DGF system, AFLGo [21]. In §8, we discuss the various aspects of CDGF and propose the direction of future improvements. We introduce the research work relevant to this paper and DGF systems in §9, and finally we conclude the paper in §10.

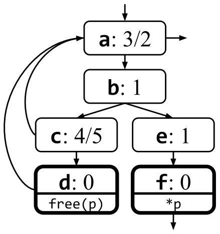


Figure 1: Example control-flow graph with DGF distances specified on each basic block.

2 Background and Motivation

In this section, we briefly introduce the fundamentals of DGF and its usage examples in §2.1 and §2.2, and point out the limitations and the consequential effects by examples in §2.3. Finally, we summarize the requirements of an augmented DGF to resolve the limitations in §2.4.

2.1 Directed Greybox Fuzzing

The *directed greybox fuzzing* (DGF) [21, 22, 44] intends to intensively fuzz a set of program locations, called *target sites*. The target sites are the preferred program locations where the seeds are driven to reach, usually set to the crash and its relevant locations. For example in Figure 1 that illustrates a simple control-flow graph with a use-after-free bug, the target sites may be set to the free location d and the use location f.

The major premise of DGF is that, when mutated, a seed *close* to the target sites is more probable to reach the target sites than the farther one. To decide the closeness of a seed, DGF first defines the distance of each basic block as the harmonic mean of the shortest path length to each target site. For example in Figure 1, the distance of a is the harmonic mean of the shortest path length to each of d and f, or $(\frac{1}{3}+\frac{1}{3})^{-1} = \frac{3}{2}$. Then, DGF calculates the distance of a given seed as the average distance of every executed basic block. For example, if the executed basic blocks are [a, b, e, f], its distance is calculated

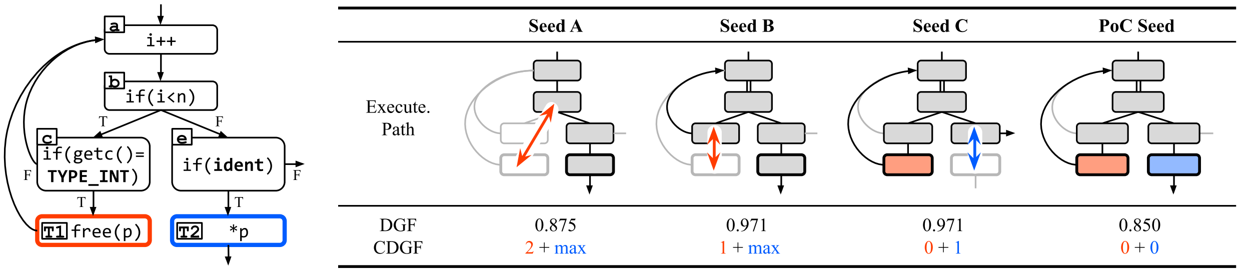


Figure 2: Simplified control-flow graph of yasm 1.3.0 (left) and the distance of example seeds in DGF and CDGF (right). To reproduce the use-after-free vulnerability, a seed must reach the red target site (`T1`) and the blue target site (`T2`) in order. CDGF distances in red and blue represent the distance portion of the target site `T1` and `T2`, respectively.

as $(\frac{3}{2}+1+1+0)/4 = 0.875$.

Generally, the seed distance gets shorter when a seed covers more target sites, but it adversely increases if it takes a longer execution path without reaching additional target sites. For example, if a seed reaches both target sites by executing $[\mathtt{a},\mathtt{b},\mathtt{c},\mathtt{d},\mathtt{a},\mathtt{b},\mathtt{e},\mathtt{f}]$, its distance is $(\frac{3}{2}+1+\frac{4}{5}+0+\frac{3}{2}+1+1+0)/8 = 0.85$, which is shorter than 0.875. On the other hands, if a seed merely takes a longer path by executing $[\mathtt{a},\mathtt{b},\mathtt{c},\mathtt{a},\mathtt{b},\mathtt{e},\mathtt{f}]$, its distance is $(\frac{3}{2}+1+\frac{4}{5}+\frac{3}{2}+1+1+0)/7 = 0.971$, which is much longer than 0.875.

2.2 Usage Example

DGF can be utilized in any use cases where the target sites can be precisely defined. Below are the prime usages where various users can leverage DGF.

2.2.1 Static Analyzer Verification

Developers of the moderate-sized projects commonly employ static analyzers, which discover the potential bug in the source code at compile time. The static analyzers provide a detailed diagnostic about the potential bug, including the crash location and the assumed data conditions for the crash.

However, developers often do not have high confidence in such diagnostics, as static analyzers are known to suffer from a high false alarm rate [27, 35]. Such diagnostics are often ignored until they are verified by the actual crash reports. Rather, developers can leverage DGF to proactively verify the diagnostics by setting the target sites to the analyzed crash locations.

2.2.2 Crash Reproduction

Developers also accept the crash reports from the users or other developers. Crash reports are often accompanied by a proof-of-concept (PoC) input and a crash dump from memory error detectors (e.g., AddressSanitizer [39] and MemorySanitizer [41]) that describes which type of crash occurs at which program location and which program locations are involved in. For example, a use-after-free crash dump specifies the location where the memory is freed.

When fixing the reported crash with only one PoC input, developers may have trouble in comprehending the crash as the PoC input represents only one concrete execution path. Furthermore, even after developers patch the source code to invalidate a given PoC input, they may not certain that the root cause has been fixed. In this situation, developers can utilize DGF to reproduce the crash by setting the target sites to the crash and its relevant locations.

2.2.3 PoC Generation

The prime targets of attackers are the outdated systems with the unpatched vulnerabilities, whose patches are already released in public. Since the patched source locations and the data conditions are supposed to fix the vulnerability, attackers can analyze the patch changelog and adversely utilize DGF to generate PoC inputs for the unpatched system, by setting the patched program locations in the pre-patched source code.

2.3 Limitation

However in practice, DGF can easily suffer from the long fuzzing time to expose the targeted crash due to the two major limitations: independent target sites and no data condition.

2.3.1 Independent Target Sites

DGF regards all target sites as *independent* and has no concept of reaching a preconditional site before a crash site. This lets DGF bypass such a precondition, precluding the chance of crash reproduction.

For example, Figure 2 describes a simplified control-flow graph of yasm 1.3.0 that suffers from a use-after-free vulnerability. To reproduce the vulnerability, a seed must have `TYPE_INT` to free the memory object at the target site `T1`, and have `ident` enabled to use the freed object at the target site `T2`. The seeds in Figure 2 are deemed as more desirable in the order of C, B and A, as the latter seeds better follow the steps required to reproduce use-after-free.

However, since DGF regards `T1` and `T2` independent, it calculates the seed distances based on the average distance to both target sites. This distance metric discourages the longer

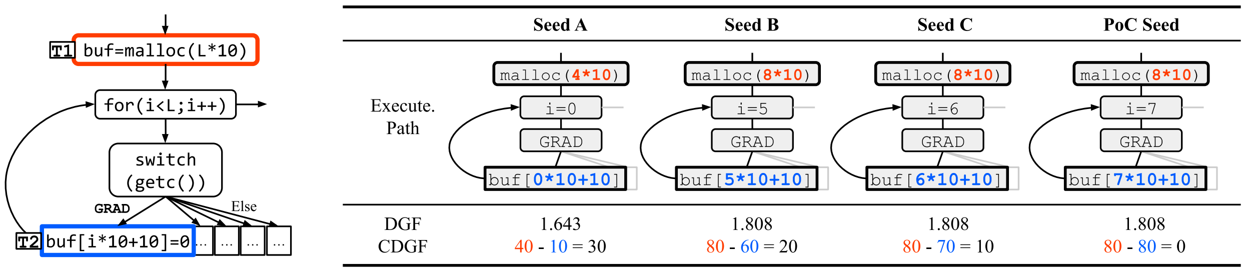


Figure 3: Simplified control-flow graph of CVE-2017-7578 (left) and the distance of example seeds in DGF and CDGF (right). To reproduce the buffer overflow vulnerability, a seed must access `buf` out of bound at the blue target site (`T2`) at the final iteration, `i=L-1`. Values in red and blue are captured at the target site `T1` and `T2`, respectively.

execution paths, which results in even longer distances in Seed B and C (0.971) than Seed A (0.875). As a result, DGF focuses on the least desirable seed, Seed A.

2.3.2 No Data Condition

DGF has no mechanism to drive the seeds to a desired *data condition*. For example, typical buffer overflow bugs are likely to occur when a seed is around the boundary of the vulnerable buffer, but DGF cannot drive a seed to such a boundary.

Figure 3 shows a simplified control-flow graph of the CVE-2017-7578 heap buffer overflow vulnerability. To reproduce the vulnerability, a seed must allocate `buf` at `T1`, and have `GRAD` to access it out of bound at `T2`, which only happens when the seed reaches `T2` at the last iteration, `i = L-1`. Each of example seeds in Figure 3 have different Ls (`L = {4,8,8}`) and access `T2` at the different iterations (`i = {0,5,6}`). The seeds are more desirable in the order of Seed C, B and A, since `GRAD` in the seed allows the program to access out of bound more closely.

However, since DGF does not recognize the data condition of the seeds, it falsely prioritizes the seeds based on their distances to the target sites. For example, DGF would prioritize Seed A the most, as it has a shortest execution path while iterating only 4 times, while others iterate 8 times.

2.4 Requirements

With the limitations in §2.3, we set two major requirements to enable the fast exposure of the targeted crash as follows.

Ordered target sites. Since most of the vulnerabilities have a separate program location that represents the precondition of the crash, DGF must be able to drive the seeds to such a location *before* the crash location.

Data conditions. Since most of the vulnerabilities are accompanied by the desired data conditions, DGF must be able to drive the seeds to such data conditions.

3 Constraint-guided DGF

In this section, we present constraint-guided directed greybox fuzzing, an augmented directed greybox fuzzing guided with a sequence of constraints. We describe a brief overview of constraint-guided directed greybox fuzzing in §3.1, and explain how constraint-guided directed greybox fuzzing can successfully prioritize the desirable seeds in §3.2.

3.1 Overview

Constraint-guided directed greybox fuzzing (CDGF) aims to satisfy a **sequence of constraints** in order, as opposed to the conventional DGF that merely aims to reach a set of independent target sites. Each constraint has its own target site that is required to be reached, and data conditions that need to be satisfied at its target site.

To achieve this goal, CDFG fuzzes in favor of the seeds that are more likely to satisfy all the constraints. In other words, it gives a shorter seed distance for following two cases: 1) if it satisfies more number of constraints and 2) if it is closer to satisfy the first unsatisfied constraint than another.

To determine how close a seed is to satisfy a sequence of given constraints, CDGF first defines the *distance of an individual constraint* as the sum of the distance to the target site and the data conditions. This yields a shorter distance when it more closely approaches its target site, and it better satisfies the data conditions. Specifically, CDGF combines the DGF-style distance for target sites [21, 22, 44] with the Angora-style distance for data conditions [23]. CDGF then defines the distance of a constraint sequence, or the *total distance*, by combining the distances of each constraint. CDGF regards the distance of an individual constraint maxed out if the preceding constraints are not satisfied, yielding a longer total distance when more constraints are left unsatisfied.

3.2 Example

3.2.1 Ordered Target Sites

Figure 4 shows the constraints for yasm 1.3.0 use-after-free in Figure 2, which can be automatically generated from the

```
CONSTRAINT %free:
  site T1
  cond "none"

CONSTRAINT %use:
  site T2
  cond "none"
```

Figure 4: Constraints to reproduce yasm 1.3.0 use-after-free.

```
CONSTRAINT %alloc:
  site T1:malloc()    # .ret = malloc(), .size = L*10
  cond "none"

CONSTRAINT %access:
  site T2:buf[]       # .addr = &buf[i*10+10]
  cond "%alloc.size <= %access.addr - %alloc.ret"
```

Figure 5: Constraints to reproduce CVE-2017-7578.

crash dump. The constraints instruct the program to first reach T1 where the memory object is freed, then to reach T2 where the freed memory object is used. Given the seeds in Figure 2 and the constraints in Figure 4, CDGF calculates the distance of the seeds as follows.

- **Seed A.** Since it approaches T1 the closest at the block b, which is two blocks away from T1, the distance of the first constraint %free is 2. Meanwhile, since it fails to reach T1, the target site of the first constraint %free, the distance of the second constraint %use is maxed out regardless of whether it reaches T2. By combining two distances, the seed distance is calculated as $2+\max$.
- **Seed B.** Since it approaches T1 closer by touching the block c, which is one block away from T1, the distance of T1 is 1. Meanwhile, the distance of %use is still maxed out, as it still fails to reach T1 before reaching T2. Combining both, the seed distance is calculated as $1+\max$.
- **Seed C.** Since it reaches T1 before T2, the distance of %free is 0. Meanwhile, since it deflects T2 by one basic block at e, the distance of %use is 1. Combining both, the seed distance is 1.

Notice that CDGF properly gives a shorter seed distance to a more desirable seed, namely $C < B < A$, where a seed with a shorter seed distance better follows the steps to reproduce use-after-free. By prioritizing the seed with a shorter distance, CDGF can develop the seeds on the basis of more desirable seeds and quickly reproduce the crash.

3.2.2 Data Conditions

Figure 5 shows the constraints for CVE-2017-7578 in Figure 3, which can also be automatically generated from the crash dump. The constraints first instruct the program to reach T1 to allocate a buffer (%alloc), and then reach T2 to access it (%access). At T2, the data condition instructs the program to reduce the difference between the size of buf and the access offset, gravitating toward the boundary (cond).

As all the seeds in Figure 3 reaches the target sites, CDGF calculates the seed distances based on the data condition distance of the second constraint %access, which is defined as the difference between the size of buf at T1 and the access offset at T2. For example, since the size of buf and the access offset in Seed A is 40 and 10 respectively, the distance of the data condition is $40-10=30$. Similarly, the distances of the data condition in Seed B and C are calculated as $80-60=20$ and $80-70=10$, respectively. Notice that how CDGF gives a shorter seed distance in the desired order, $C < B < A$, where the seeds better drive the access offset out of bound.

4 Constraints

To enable CDGF, it is essential to define the constraint itself and the distance metric of the constraints. In this section, we first define the constraint in §4.1, then define the distance of a sequence of constraints in §4.2.

4.1 Definition

A constraint is a combination of a *target site* to reach, and the *data conditions* to satisfy at the target site. A constraint is called satisfied if: i) the program reaches the target site, and ii) the data conditions are all satisfied at the target site, if any. Examples are the CONSTRAINT %free, %use in Figure 4 and %alloc and %access in Figure 5.

Variable capturing. Once the target site is reached, the constraint captures the variables used at the target site. Which variables are captured depends on the type of the location indicated by the target site. For example in Figure 5, the dereferenced address (&buf[i*10+10]) is captured as addr because T2:buf[] is a dereference, and the size of the allocation (L*10) and the allocated address are captured as size and ret respectively, because T1:malloc() is an allocation.

Data condition. A data condition is a boolean expression between captured variables and a comparison operator that needs to be satisfied at the target site. A data condition can reference any variables captured by the preceding constraints or the constraint that it belongs to. For example in Figure 5, **cond** "%alloc.size <= %access.addr - %alloc.ret" is the data condition that references the variables size and ret from the constraint %alloc and the variable addr from the constraint %access.

Orderedness. Constraints may be specified more than one. In such cases, they must be satisfied in the specified order. For example in Figure 4, the %free constraint must be satisfied before the %use constraint to trigger use-after-free.

4.2 Distance of Constraints

In this section, we develop the distance of a constraint sequence called *total distance*, given a basic block trace that a seed executes at runtime. To do so, we first define the distance to a target site in §4.2.1, and define the distance of data conditions in §4.2.2. We combine two distances to define the distance of a constraint in §4.2.3, and finally we define the total distance in §4.2.4.

4.2.1 Target Site Distance

Basic block distance. We define the distance to a target site in a similar manner to the prior work [21, 22, 44]. First, the distance of two basic blocks, or the *basic block distance* $d(B_1, B_2)$ is defined as

$$d(B_1, B_2) = \begin{cases} 0 & \text{if } B_1 = B_2, \\ \min(d(B_s, B_2) + 1), \forall B_s & \text{if } B_1 \rightsquigarrow B_2, \\ \infty & \text{else,} \end{cases} \qquad (1)$$

where B_s is the successor basic blocks of B_1, including the called basic blocks if B_1 contains any call instruction. We denote by $B_1 \rightsquigarrow B_2$ if B_2 is reachable from B_1. We take the minimum distance if two basic blocks are reachable via multiple control-flow paths.

Target site distance. Using the basic block distance, we define the distance to a target site as the basic block distance between its target site and the current basic block. Specifically, let $\vec{B} = [B^1, B^2, \ldots]$ be a basic block trace, a sequence of executed basic blocks, and B^* be the target site. Then, when the program executes B^n, the distance of a target site $D_{\texttt{TARGET}}$ is defined as

$$D_{\texttt{TARGET}} = d(B^n, B^*). \qquad (2)$$

From now on, we decorate an arbitrary variable $\bigcirc$ with $\bigcirc^n$ to denote the value of the variable *at the moment* when the program executes B^n in the trace. For example, $D^n_{\texttt{TARGET}}$ denotes the target site distance when the program executes the nth basic block in the trace, B^n.

4.2.2 Data Condition Distance

Distance of an individual data condition. The distance of an individual data condition is based on the distance of integer values $\hat{d}(\vec{n})$. Table 1 shows the definition of $\hat{d}_\Box(\vec{n})$ with a given comparison operator, which is similar to the definition of [23]. Basically, the distance between two integers is 0 if the comparison is true, and farther from 0 as it is farther from the solution. One additional operator with three integer values $n_1 <= n_2 < n_3$ yields 0 when n_2 is between the two integers n_1 and n_3, and produces a larger value as n_2 is further from the range $[n_1, n_3)$. We assume $\hat{d}(\vec{n})$ is ∞ if any one of integers in $\vec{n}$ is undefined.

Expression	Meaning	$\hat{d}(\vec{n})$
$n_1 == n_2$	Equal	$\lvert n_1 - n_2 \rvert$
$n_1 \,!= n_2$	Not equal	0 if true, otherwise 1.
$n_1 >= n_2$	Greater than or equal to	$\max(n_2 - n_1, 0)$
$n_1 > n_2$	Greater than	$\max(n_2 - n_1 + 1, 0)$
$n_1 <= n_2$	Less than or equal to	$\max(n_1 - n_2, 0)$
$n_1 < n_2$	Less than	$\max(n_1 - n_2 + 1, 0)$
$n_1 <= n_2 < n_3$	Between	$\max(n_1 - n_2, n_2 - n_3 + 1, 0)$

Table 1: The distance $\hat{d}(\vec{n})$ of integer values.

Using the distance of integer values $\hat{d}(\vec{n})$ and given a data condition Q, we define the distance of a data condition $\hat{d}^n(Q)$ when the program executes the nth basic block B^n as

$$\hat{d}^n(Q) = \min(\hat{d}_\Box(\vec{n})), \forall \vec{n} \in Var^n(Q), \qquad (3)$$

where $Var^n(Q)$ is a set of variable vectors captured until the program executes B^n, and $\Box$ is the comparison operator of Q. Basically, $\hat{d}^n(Q)$ is equal to the minimum distance of all captured variables until B^n, or ∞ if any one of the variables is not captured yet, thus undefined.

As a special case, if a condition is denoted by `assert` rather than `cond`, we define the distance of such a data condition in a pass-or-fail manner, yielding 0 when the condition is satisfied or ∞ otherwise. This prevents falsely measuring the distance when the numerical aspects of the integer values bear no significance, such as the pointer addresses.

Distance of multiple data conditions. The distance of multiple data conditions is supposed to indicate how close a seed is to satisfy all the data conditions. To this end, we define the distance of data conditions so that it gets shorter when more data conditions are satisfied, and if the number of satisfied data conditions is the same, the first unsatisfied data condition is closer to be satisfied. Let $\vec{Q} = [Q_1, Q_2, \ldots]$ be a sequence of data conditions, and ρ is the index of the first unsatisfied data condition.[1] Then the distance of data conditions $D^n_{\texttt{DATA}}$ can be defined as

$$D^n_{\texttt{DATA}} = c_{data} \cdot N_{unsat} + \min(c_{data}, \hat{d}^n(Q_\rho)), \qquad (4)$$

where $N_{unsat} = N(\vec{Q}) - \rho$ is the number of unsatisfied data conditions, and c_{data} is the maximum distance of an individual data condition. We set c_{data} to 2^{32}, so that a single data condition distance can represent any distance values between 4-byte integers. Notice that $D^n_{\texttt{DATA}}$ gives a shorter distance when N_{unsat} gets smaller, and the distance of the first unsatisfied data condition $\hat{d}(Q_\rho)$ has a shorter distance.

4.2.3 Constraint Distance

The distance of an individual constraint is the sum of the target site distance and the data condition distance. Formally,

[1] We assume ρ is the last index if all data conditions are satisfied.

if $D^n_{\texttt{CONSTR}}$ is the distance of a constraint,

$$D^n = D^n_{\texttt{TARGET}} + D^n_{\texttt{DATA}}. \tag{5}$$

Until the constraint is satisfied, D^n changes the value in the following ways.

1. *Before the target site:* $D^n = d(B^n, B^*) + c_{data} \cdot N(\vec{Q})$.

Before reaching the target site B^*, the distance to the target site is $d(B^n, B^*)$ when the program executes B^n in the trace. Meanwhile, the distance of the data conditions is at its maximum, $c_{data} \cdot N(\vec{Q})$, because not all referenced variables are captured (i.e., defined) until the program reaches the target site.

2. *At the target site:* $D^n = 0 + D^n_{\texttt{DATA}}$.

After reaching the target site, the distance of a constraint is solely determined by the distance of its data conditions, $D^n_{\texttt{DATA}}$. The distance gets shorter as more data conditions are satisfied and the first unsatisfied data conditions is in a closer condition to be satisfied.

3. *When constraint satisfied:* $D^n = 0$.

Similar to other distance definitions, the distance of a constraint is 0 if the constraint is satisfied, that is when: i) the target site is reached and ii) its data conditions are all satisfied at the target site. This is naturally derived from the other distance definitions, because such situation indicates both $D^n_{\texttt{TARGET}}$ and $D^n_{\texttt{DATA}}$ are 0, so is the sum of them.

4.2.4 Total Distance

The distance of a constraint sequence, or the *total distance*, is the serial combination of the distance of each individual constraint D^n_i. Let $\vec{B^*} = [B^*_1, B^*_2, ..., B^*_M]$ be a sequence of target sites that belong to each of the M constraints. Furthermore, let τ^n be the index of the first unsatisfied constraint.[2] Then the total distance $\mathbf{D}^n$ can be defined as

$$\mathbf{D}^n = c_{con} \cdot (N(\vec{B^*}) - \tau^n) + \min(c_{con}, D^n_{\tau^n}). \tag{6}$$

Here, c_{con} is the maximum distance of an individual constraint, and $N(\vec{B^*})$ is the number of target sites in $\vec{B^*}$. We set c_{con} to 2^{35}, so that a constraint can accommodate up to 8 data conditions. Until all constraints are satisfied, $\mathbf{D}^n$ changes the value in the following ways.

1. *When no constraint satisfied:* $\mathbf{D}^n = c_{con} \cdot (N(\vec{B^*}) - 1) + \min(c_{con}, D^n_1)$.

Since no constraint has been satisfied yet, τ^n is 1 because the first unsatisfied constraint is the very first constraint. Hence, the total distance is the distance of the first constraint, plus the maximum distances of the rest of the constraints.

```
CONSTRAINT %cause:
  site <cause_site>
  cond "none"

CONSTRAINT %trans:
  site <trans_site>
  cond "none"

CONSTRAINT %crash:
  site <crash_site>
  cond "none"
```

Figure 6: nT constraint template.

2. *When last constraint remains:* $\mathbf{D}^n = \min(c_{con}, D^n_M)$.

Once all constraints are satisfied except the last one, the total distance is solely determined by the distance of the last constraint, because $(N(\vec{B^*}) - \tau^n) = (M - M) = 0$.

3. *When all constraints satisfied:* $\mathbf{D}^n = 0$.

Similar to the distances defined so far, the total distance gets zero as both of the terms are reduced to zero. Inversely, the zero total distance indicates that all constraints are satisfied in the specified order.

Finally, we define the total distance of a seed as the minimum total distance throughout its execution. Formally, if s is a seed that generates the basic block trace $\vec{B}$, the total distance of a seed $\mathbf{D}(s)$ is defined as $\mathbf{D}(s) = \min(\mathbf{D}^n), \forall n$.

5 Constraint Generation

The basic approach of constraint generation is, given an additional information source, finding proper target sites and data conditions to fill out a pre-defined constraint template. We design constraint generation for two such sources: crash dumps from memory error detectors [39, 41] and patch changelogs.

5.1 Crash Dump

Constraint generation for crash dumps refers to the bug types to choose an appropriate template. We compose three templates that can support seven bug types in total. In particular, the **nT** template with multiple target sites handles use-after-free, double-free, and use-of-uninitialized-value (§5.1.1), the **2T+D** template with two target sites and data conditions handles stack-buffer-overflow and heap-buffer-overflow (§5.1.2), and the **1T+D** template with one target site and data conditions handles assertion-failure and divide-by-zero (§5.1.3).

5.1.1 Multiple Target Sites (nT)

Figure 6 shows the nT constraint template with multiple target sites. The template is useful when a crash dump informs the target sites required to be reached in order. The bracket-enclosed placeholders are replaced to the program location found at the top of the corresponding stack dumps.

[2] Similar to ρ, we assume τ^n is M if all constraints are satisfied.

```
CONSTRAINT %alloc:
  site <alloc_site>
  cond "none"

CONSTRAINT %access:
  site <access_site>
  assert "%alloc.ret <= %access.addr < %alloc.endaddr"
  cond "%alloc.endaddr <= %access.addr"
```

Figure 7: `2T+D` constraint template with buffer overflow data conditions. `%alloc.endaddr` is `%alloc.ret + %alloc.size`.

```
CONSTRAINT %constr:
  site <target_site>
  cond "<data_cond>"
```

Figure 8: `1T+D` constraint template.

Avoiding wrapper functions. To make the target sites more representative, we avoid choosing a target site inside memory wrappers by checking if the name of the stack frame caller contains the keywords such as "alloc", "free", or "mem". If a location is inside a memory wrapper, we take the location of lower stack frames instead.

Constraint description. The template specifies multiple target sites that are required to be reached in the specified order to reproduce the crash. The `%cause` constraint represents where the cause is generated. The `%trans` constraint represents where the cause is transferred. Finally, the `%crash` constraint represents where the crash occurs.

Corresponding bug types. *Use-after-free*, *double-free*, and *use-of-uninitialized-value* correspond to the `nT` constraint template. Use-after-free and double-free bugs set <`cause_site`> and <`crash_site`> to where an address is freed and used, and do not use the `%trans` constraint. Use-of-uninitialized-value sets <`cause_site`> and <`crash_site`> to where the uninitialized value is created and used, and set <`trans_site`> to where it is transferred if the uninitialized value is mediated by multiple variables.

5.1.2 Two Target Sites with Data Conditions (2T+D)

Figure 7 shows the `2T+D` constraint template with two target sites and data conditions. The illustrated data conditions are for buffer-overflow bugs, where `endaddr` denotes the end of the allocated memory, namely `ret + size`. Similar to the `nT` template, the bracket-enclosed placeholders are replaced to the program location at the top of the corresponding stack dumps. We capture the variables inside of memory wrappers, even if the target sites are set to outside of them.

Constraint description. The template specifies two target sites and data conditions to reproduce buffer-overflow. The `%alloc` constraint first guides the program to where a buffer is allocated. When <`alloc_site`> is reached, it captures the begin and end address of the allocated buffer as `ret` and `endaddr`. Next, the `%access` constraint guides to where the buffer is accessed. When <`access_site`> is reached, the `assert` condition first identifies whether the accessed address (`%access.addr`) belongs to the buffer allocated by <`alloc_site`>. If it does, to increase the likelihood of overflow, the `cond` condition drives the accessed address to the boundary.

Notice that the data conditions are intended to *drive* buffer-overflow, not to *detect* one; while the likelihood of buffer-overflow is increased by both data conditions (i.e., `assert` and `cond`), the actual buffer-overflow will be detected by the memory error detectors [39, 41] regardless of whether both data conditions are satisfied.

Corresponding bug types. *Heap-buffer-overflow* and *stack-buffer-overflow* correspond to the `2T+D` constraint template. In both bugs, <`alloc_site`> is set to where a buffer is allocated and <`access_site`> to where a buffer is accessed out of bound. The template currently does not support global-buffer-overflow and buffer-underflow. See §8 for details.

5.1.3 One Target Site with Data Conditions (1T+D)

Figure 8 shows the `1T+D` constraint template with one target site and data conditions. The template is useful when a crash dump does not reveal multiple target sites, but the bug type suggests some definite buggy conditions.

Constraint description. The template specifies one target site that is required to be reached and data conditions that needs to be satisfied at the target site. The `%constr` constraint specifies such a target site (<`target_site`>) and data conditions (<`data_cond`>).

Corresponding bug types. *Divide-by-zero* and *assertion-failure* correspond to the `1T+D` template. Divide-by-zero sets <`target_site`> to the crashing division expression and <`data_cond`> to `%constr.rhs == 0`, where `rhs` indicates the divisor operand. Assertion-failure sets <`target_site`> to the failed assertion check, and <`data_cond`> to the negated assertion condition.

5.2 Patch Changelog

Constraint generation for patch changelogs uses the `1T+D` constraint template shown in Figure 8. Since the patch is supposed to fix the cause of the crash, we utilize the changed source locations by assuming that they signify the cause.

Constraint description. The constraint first guides the program to <`target_site`> that represents the changed locations. When <`target_site`> is reached, it attempts to generate the cause by satisfying <`data_cond`> that represents the changed data conditions.

Determining constraint. To find a proper constraint, a given patch changelog is matched with a series of pre-defined cases, earlier cases being potentially more direct to the cause of the bug. The following describe the case matching, which is algorithmically described in Appendix C.

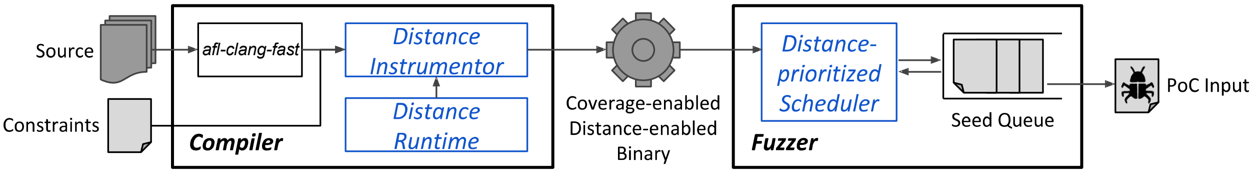


Figure 9: System overview of CAFL.

- `C1.` If any new exception checks are introduced, it sets `<target_site>` to their source locations and creates `<data_cond>` with newly introduced exception conditions. We assume the conditions that lead to a return statement or a function call with a keyword such as "throw" or "error" suggest exception checks.
- `C2.` If any branch condition is changed, it sets `<target_site>` to the changed conditions and creates `<data_cond>` where the pre- and post-patched conditions are mutually exclusive to each other. In other words, if C_{pre} and C_{post} are the pre- and post-patch conditions, `<data_cond>` = (C_{pre} && !C_{post}) || (!C_{pre} && C_{post}).
- `C3.` If any variables are replaced, it sets `<target_site>` to the replaced variable and creates `<data_cond>` that tests if the value of the pre-patched variable is not equal to the post-patched one.
- `C4.` If all the preceding cases are not applicable, it falls back to the data-condition-free constraint, setting `<target_site>` to all the changed program locations.

Multiple target sites. If the changed locations are more than one, it ties all changed locations with a sentinel function that represents a single unified target site, and sets `<target_site>` to the sentinel function. Specifically, it inserts a sentinel function call to each of change locations, so that the program calls the sentinel function whenever it reaches them.

6 Implementation

We implemented CAFL, the prototype fuzzing system of CDGF based on AFL 2.52b [46]. In this section, we first describe a brief system overview in §6.1, and explain the operation of the CAFL components in §6.2 to §6.4.

6.1 System Overview

Figure 9 shows the system overview of CAFL. First, the CAFL compiler accepts the source code and the constraints such as Figure 4, and instruments both edge coverage and the target site distances. The target site distances are instrumented by installing the *checkpoint* API calls provided by the CAFL runtime. Then, the CAFL fuzzer fuzzes the binary and receives the seed distance from the CAFL runtime, prioritizing the seeds with shorter total distances.

6.2 CAFL Compiler

Coverage instrumentation. The CAFL compiler generates the LLVM [17, 28] IR bitcode and annotates the target sites to prevent them optimized out. It then instruments the edge coverage using the AFL [46] instrumentation compiler with optimizations enabled.

Call graph construction. To calculate the target site distance, the CAFL compiler first constructs the program-wide call graph. When it comes to the function pointers, the CAFL compiler assumes all the functions whose prototypes are exactly matching as the potential callees. If such a function is not found, the CAFL compiler alternatively assumes the functions with partially matching prototypes at the earlier part as the potential callees.

Target site distance instrumentation. Starting from each target site, the CAFL compiler calculates the target site distance of the basic blocks and inserts the *checkpoint* calls, while recursively crawling up the control-flow graph and the call graph until it reaches the `main` function. As a special case, the CAFL compiler attaches the captured variables to the checkpoint call at the target site and forwards them to the CAFL runtime.

6.3 CAFL Runtime

Seed distance tracking. At fuzzing time, the CAFL runtime keeps track of the seed distance using the target site distance feedback through the checkpoints. The instrumented binary forwards the tuple of $[\tau, d(B^n, B^*_\tau)]$ through the checkpoints, where τ is the index of the constraint and $d(B^n, B^*_\tau)$ is the target site distance of the τth constraint. The CAFL runtime selectively accepts the target site distance of the first unsatisfied constraint, and updates the current seed distance.

At the target site, the CAFL runtime receives the captured variables through the checkpoint call and calculates the data condition distances. To prevent the released memory from disrupting distance measurement, the CAFL runtime also disposes a captured variable if i) it is a memory pointer and ii) released by `free`/`realloc` (heap objects) or stack unwinding (stack objects). If the distance of the current constraint gets 0, the CAFL runtime advances to the next constraint.

Seed distance reporting. While tracking the seed distance, the CAFL runtime reports the distance of the current seed to the CAFL fuzzer via a dedicated shared memory interface.

```
CONSTRAINT %cause:
  site <cause_site>
  cond "none"

CONSTRAINT %crash:
  site <crash_site>:<crash_var>
  cond "<crash_begin> <= %crash.value < <crash_end>"
```

Figure 10: Constraint template for LAVA-1 crashes.

To facilitate monitoring the fuzzing status, the CAFL runtime also reports additional runtime statistics, such as at which constraint a seed is stuck.

6.4 CAFL Fuzzer

Seed scoring. As the seeds with shorter total distances are generally deemed as desirable, the CAFL fuzzer first scores each seed negative-proportionally to its total distance, giving a bigger score to a shorter total distance. Meanwhile, some may be the local minima whose total distance cannot be shortened further. To avoid such local minimum seeds, the CAFL fuzzer exponentially scales down the seed score with respect to the fuzzed times and the stuck depth, the seed depth during which a seed fails to reduce the total distance shorter than the shortest seed distance of all the parent seeds. Formally, given a seed s with the total distance of $\mathbf{D}(s)$, the score of the seed $\mathbf{S}(s)$ is

$$\begin{aligned}\mathbf{S}(s) \quad = \quad & (\mathbf{D}_{max} - \mathbf{D}(s)) \cdot \mathrm{pow}(c_{fuzz}, NumFuzzed(s)) \cdot \\ & \mathrm{pow}(c_{stuck}, StuckDepth(s)), \end{aligned} \qquad (7)$$

where $\mathbf{D}_{max} = c_{con} \cdot N(\vec{B^*})$ is the maximum total distance, and $NumFuzzed(s)$ and $StuckDepth(s)$ is the fuzzed times and the stuck depth of the seed s, respectively. c_{fuzz} and c_{stuck} are the scale-down factors, which we set to 0.95 and 0.85.

Seed creation. The CAFL fuzzer creates a new seed whenever it observes a seed whose score is bigger than the current biggest. The CAFL fuzzer also creates seeds in a conventional way, namely when a seed covers new control-flow edges, to diversify the data context of seeds.

Seed prioritization. The CAFL fuzzer modifies the seed scheduling algorithm of AFL by regulating the selection probability of each seed based on its score. Specifically, the CAFL fuzzer ranks each seed in an increasing order of its score, and gives an exponentially higher probability of being chosen. We give the probabilities with respect to the ranks rather than the scores, as the total distance is a combination of two different distance metrics whose numerical scales are not compatible. Formally, if $R(s)$ is the rank of the seed s, the probability of choosing the seed s is defined as $\mathbf{P}(s) = 1/\exp(R(s))$.

7 Evaluation

Since the state-of-the-art DGF system Hawkeye [22] is not publicly available, we compare CAFL with AFLGo [21] that Hawkeye is built upon. We evaluate both on a server node with 20-core Intel Xeon Gold 6209U CPU @ 2.10GHz and 502 GB of DDR4 main memory. We run CAFL on Ubuntu 18.04 and AFLGo on Ubuntu 16.04, due to the OS compatibility issue in AFLGo. We repeat each evaluation 3 times and average them, except when AFLGo exceeds the timeout. We configure AFLGo with the default parameters described in the official repository (`-z exp -c 45m`) [1].

In this section, we first present the microbenchmark results using LAVA-1 [25] in §7.1, and present the crash reproduction time upon 47 real-world crashes in §7.2. Finally, we present the PoC generation time upon 12 crashes in §7.3. All constraints are automatically generated.

ID	Buggy Range Size	AFLGo	CAFL	
			2T	2T+D
4961	0x10000000	>1000.0 m	0.9 m	0.2 m
7002	0x10000000	4.3 m	>1000.0 m	0.1 m
13796	0x10000000	>1000.0 m	459.3 m	121.5 m
292	0x200000	>1000.0 m	>1000.0 m	0.6 m
660	0x200000	>1000.0 m	>1000.0 m	0.6 m
3089	0x200000	>1000.0 m	>1000.0 m	0.8 m
4383	0x200000	>1000.0 m	>1000.0 m	0.2 m
7700	0x200000	>1000.0 m	>1000.0 m	0.3 m
14324	0x200000	>1000.0 m	>1000.0 m	0.9 m
2543	0x4000	>1000.0 m	>1000.0 m	0.3 m
4049	0x4000	>1000.0 m	>1000.0 m	0.6 m
1199	0x80	>1000.0 m	>1000.0 m	0.3 m
2285	0x80	>1000.0 m	>1000.0 m	0.7 m
9763	0x80	>1000.0 m	>1000.0 m	0.3 m
16689	0x80	>1000.0 m	>1000.0 m	0.7 m
17222	0x80	>1000.0 m	>1000.0 m	1.2 m
357	0x1	>1000.0 m	>1000.0 m	1.3 m
3377	0x1	>1000.0 m	>1000.0 m	0.7 m

Table 2: LAVA-1 crash reproduction time comparison. All reproduction times are cut off at 1000 minutes. **T** stands for **t**arget sites, and **D** stands for **d**ata conditions.

7.1 Microbenchmark: LAVA-1

We compare the crash reproduction time with 18 crashes from LAVA-1 [25] by measuring the time taken to reproduce the crash injected to the Linux `file` command [8]. Figure 10 shows the constraint template used to reproduce the crashes in LAVA-1. Since LAVA-1 provides the detail crash information including i) the program location involved in the cause of the crash, ii) the program location where the program crashes, and iii) the buggy variable and its data range where it causes the crash, we utilize the information to fill the placeholders in the constraint template. Specifically, we set the cause location and the crash location to `<cause_site>` and `<crash_site>` respectively, and the begin and end value of the buggy range to `<crash_begin>` and `<crash_end>`. Setting this constraints as **2T+D**, we construct the constraints for **2T** by disabling the data condition to `"none"`. We set the targets sites for AFLGo to the same as in **2T**.

Program	Bug Location	Bug Type	Template	AFLGo		CAFL			Speedup
				1T	nT	1T	nT	nT+D	
gifsicle 1.90	fmalloc.c:19	Double free	nT	8.0 m	7.0 m	15.2 m	7.2 m	-	1.0x
gifsicle 1.90	giffunc.c:185	Use after free	nT	15.2 m	27.6 m	27.0 m	18.4 m	-	0.8x
ImageM 7.0.6-5	mat.c:1374	Use after free	nT	180.1 m	404.5 m	17.3 m	8.6 m	-	20.9x
libming 0.4.8	decompile.c:398	Use after free	nT	103.8 m	33.0 m	12.7 m	14.8 m	-	2.6x
libtiff 4.0.3	tiff2pdf.c:394	Use after free	nT	>6000.0 m	>6000.0 m	>6000.0 m	1593.5 m	-	3.8x
libtiff 4.0.9	tiff2pdf.c:405	Use after free	nT	>6000.0 m	>6000.0 m	>6000.0 m	2887.0 m	-	2.1x
libzip 1.2.0	zip_buffer.c:53	Use after free	nT	>2000.0 m	379.5 m	1642.3 m	825.3 m	-	0.5x
mJS 1.21	mjs_string.c:524	Use after free	nT	32.4 m	58.8 m	31.1 m	24.2 m	-	1.3x
nasm 2.14rc0	preproc.c:1290	Use after free	nT	*	*	774.2 m	65.2 m	-	-
nasm 2.14rc16	preproc.c:5055	Use after free	nT	*	*	7176.1 m	>12000.0 m	-	-
yasm 1.3.0	intnum.c:415	Use after free	nT	>2000.0 m	218.2 m	120.2 m	32.3 m	-	6.8x
jbig2dec 0.16	jbig2_arith.c:264	Uninitialized value	nT	168.9 m	53.3 m	130.1 m	56.0 m	-	1.0x
jbig2dec 0.16	jbig2_mmr.c:88	Uninitialized value	nT	>2000.0 m	>2000.0 m	1064.2 m	710.3 m	-	2.8x
jasper 1.900.12	jas_seq.c:90	Assertion failure	1T+D	2.4 m	-	3.2 m	-	1.2 m	2.0x
jasper 1.900.13	jpc_dec.c:1817	Assertion failure	1T+D	136.1 m	-	30.0 m	-	15.7 m	8.7x
jasper 1.900.13	jpc_bs.c:197	Assertion failure	1T+D	16.7 m	-	6.1 m	-	9.9 m	2.7x
jasper 1.900.13	jpc_t2cod.c:297	Assertion failure	1T+D	>2000.0 m	-	1990.1 m	-	281.8 m	7.1x
jasper 1.900.17	jpc_math.c:94	Assertion failure	1T+D	130.1 m	-	220.1 m	-	122.4 m	1.1x
libsixel 1.8.3	stb_image.h:5052	Assertion failure	1T+D	*	-	15.4 m	-	31.9 m	-
libtiff 4.0.7	tif_dirwrite.c:2098	Assertion failure	1T+D	58.1 m	-	165.3 m	-	24.4 m	2.4x
GraphicsM 1.3.28	png.c:4638	Divide by zero	1T+D	*	-	4.5 m	-	1.9 m	-
imagew 1.3.1	imagew-cmd.c:850	Divide by zero	1T+D	1.0 m	-	2.6 m	-	1.4 m	0.7x
lame 3.99.5	get_audio.c:1454	Divide by zero	1T+D	3.4 m	-	2.7 m	-	1.5 m	2.3x
libtiff 4.0.7	tif_read.c:351	Divide by zero	1T+D	>9000.0 m	-	>9000.0 m	-	2046.0 m	4.4x
Average									nT: 2.12x / 1T+D: 2.63x

Table 3: Crash reproduction time comparison with nT and 1T+D. All reproduction times are cut off at 2000 minutes, except libtiff use-after-free (6000 minuntes) and divide-by-zero (9000 minutes), and nasm 2.14rc16 use-after-free (12000 minutes) due to the long reproduction time. Speedup is between the shortest times of AFLGo and CAFL. Underlined times are the shortest. -: Not applicable. *: AFLGo fails to launch.

Table 2 shows the crash reproduction time in AFLGo and CAFL with two different constraint settings. The crashes are sorted in the order of the buggy range size, the higher the wider. AFLGo and CAFL with **2T** all fails to reproduce the crashes until the timeout when the buggy data range is narrower than 0x10000000 = 2^{27}, but CAFL with **2T+D** successfully reproduces the crashes in less than 2 minutes, except the crash 13796 where reaching the crash location takes most of the fuzzing time. It is worth noting that AFLGo and CAFL with **2T** cannot reproduce the crash 4961 and 7002 respectively, even if the crashes have the widest buggy data range. This is because both do not recognize the buggy data condition and ignore the seeds that are in a close condition, letting them hidden among the irrelevant seeds.

7.2 Crash Reproduction

We compare the crash reproduction time with 47 crashes from various real-world programs by measuring the time taken to generate the same kind of crash at the same crash site. We set the timeout to 2000 minutes, except four crashes that require a longer timeout due to the long reproduction time. All constraints are automatically generated with AddressSanitizer [39] and MemorySanitizer [41] crash dumps.

To demonstrate the effect of the ordered target sites and the data conditions, we measure the crash reproduction time with various constraint settings. In **1T**, we only set the crash location to a target site. In **nT**, we set all available target sites without data conditions. This setting corresponds to the nT template, and the 1T+D and 2T+D templates without data conditions. In **nT+D**, we set all available target sites *with* data conditions. This setting corresponds to the 1T+D and 2T+D templates. We set the same target sites for AFLGo.

Table 3 shows the crash reproduction times with the nT and 1T+D templates. With the nT template, CAFL outperforms AFLGo by 2.12x on average. CAFL with **nT** generally performs better for most crashes, but is merely effective when the crash is relatively common (gifsicle 1.90 and mJS 1.21) or the cause site always comes with the crash site (libzip 1.20 and jbig2dec 0.16). The crash in nasm 2.14rc16 requires some grammar knowledge, which can be easily supported in cooperation with grammar fuzzing [19, 20] in the future.

With the 1T+D template, CAFL outperforms AFLGo by 2.63x on average. Notice that the data-condition-enabled **nT+D** setting constantly shows superior performance over AFLGo in most crashes, and even when the data conditions are not effective, CAFL with **nT+D** shows at least tentative performance compared to AFLGo.

Table 4 shows the crash reproduction times with the 2T+D template, where CAFL outperforms AFLGo by 3.65x on average. Notice that the data-condition-free **nT** setting sometimes takes longer reproduction time than **1T** in both AFLGo and CAFL. This is mainly because the allocation site is commonly reached regardless of it is targeted or not, causing nothing but additional overheads in seed scheduling. However in CAFL, this performance degradation is constantly compensated by the data conditions represented by **nT+D**. Overall, CAFL outperforms AFLGo by 2.88x on average. See Appendix D for the detailed analysis on the minimum distance change during the fuzzing session on various constraint settings.

Program	Bug Location	Bug Type	Template	AFLGo		CAFL			Speedup
				1T	nT	1T	nT	nT+D	
jasper 1.900.22	jpc_tsfb.c:225	Stack buffer overflow	2T+D	116.3 m	†	98.0 m	†	34.2 m	3.4x
lame 3.99.5	get_audio.c:1205	Stack buffer overflow	2T+D	142.1 m	†	33.8 m	†	34.2 m	4.2x
libsixel 1.8.1	frompnm.c:144	Stack buffer overflow	2T+D	*	*	33.3 m	†	34.0 m	-
fig2dev 3.2.7b	gensvg.c:1005	Heap buffer overflow	2T+D	56.2 m	37.7 m	31.0 m	22.7 m	35.0 m	1.7x
fig2dev 3.2.7b	read.c:1532	Heap buffer overflow	2T+D	180.9 m	151.5 m	16.4 m	11.0 m	14.9 m	13.8x
GraphicsM 1.4	pict.c:1114	Heap buffer overflow	2T+D	>2000.0 m	>2000.0 m	>2000.0 m	>2000.0 m	198.6 m	10.1x
GraphicsM 1.4	sun.c:223	Heap buffer overflow	2T+D	36.1 m	19.1 m	77.4 m	148.5 m	32.6 m	0.6x
GraphicsM 1.4	miff.c:428	Heap buffer overflow	2T+D	*	*	2.4 m	1.6 m	2.0 m	-
ImageM 7.0.3-6	pixel-accessor.h:507	Heap buffer overflow	2T+D	*	*	22.5 m	25.0 m	13.5 m	-
jbig2dec 0.16	jbig2_generic.c:356	Heap buffer overflow	2T+D	19.1 m	20.9 m	9.0 m	8.0 m	6.5 m	2.9x
jasper 1.900.3	jpc_dec.c:1668	Heap buffer overflow	2T+D	61.9 m	79.9 m	2.2 m	3.1 m	1.4 m	44.2x
libming 0.4.7	parser.c:66	Heap buffer overflow	2T+D	81.8 m	620.4 m	124.1 m	27.3 m	15.2 m	5.4x
libsixel 1.8.2	frompnm.c:289	Heap buffer overflow	2T+D	*	*	40.5 m	202.0 m	61.3 m	-
libsixel 1.8.4	fromgif.c:278	Heap buffer overflow	2T+D	*	*	63.6 m	81.2 m	14.6 m	-
libarchive 3.1.2	format_zip.c:694	Heap buffer overflow	2T+D	>2000.0 m	>2000.0 m	1002.3 m	‡	690.9 m	2.9x
libtiff 4.0.6	tif_packbits.c:85	Heap buffer overflow	2T+D	2.2 m	1.9 m	5.0 m	0.8 m	1.0 m	2.4x
libtiff 4.0.7	tif_swab.c:291	Heap buffer overflow	2T+D	>2000.0 m	>2000.0 m	645.6 m	1560.3 m	213.7 m	9.4x
libtiff 4.0.7	tif_unix.c:115	Heap buffer overflow	2T+D	74.3 m	154.4 m	280.8 m	137.2 m	27.8 m	2.7x
libtiff 4.0.7	tiffcrop.c:3911	Heap buffer overflow	2T+D	7.8 m	22.8 m	8.6 m	35.3 m	23.3 m	0.9x
libtiff 4.0.7	tiff2ps.c:2487	Heap buffer overflow	2T+D	2.0 m	1.6 m	4.5 m	2.5 m	5.8 m	0.6x
libtiff 4.0.9	pal2rgb.c:196	Heap buffer overflow	2T+D	5.3 m	37.0 m	3.7 m	11.2 m	10.2 m	1.4x
libtiff 4.0.10	tiff2ps.c:2479	Heap buffer overflow	2T+D	283.0 m	65.5 m	50.7 m	10.8 m	7.7 m	8.5x
mJS 1.21	mjs_string.c:58	Heap buffer overflow	2T+D	229.8 m	39.5 m	86.2 m	31.0 m	5.0 m	7.9x
Average									3.65x

Table 4: Crash reproduction time comparison with 2T+D. All reproduction times are cut off at 2000 minutes. Speedup is between the shortest times of AFLGo and CAFL. Underlined times are the shortest. -: Not applicable. *: AFLGo fails to launch. †: Essentially equivalent to **1T**. ‡: Skipped due to the high instrumentation overheads.

7.3 PoC Generation

We compare the PoC generation time with 12 crashes. Similar to §7.2, all constraints are automatically generated from the patch changelog from git [9] and Mercurial [16].

We measure the PoC generation time with various constraint settings. In **T**, we set the target site to the sentinel function, which ties the selected source locations as described in §5.2. In **T+D**, we add the data condition created by the case matching in §5.2. We set the same target sites for AFLGo.

Table 5 shows the PoC generation time in AFLGo and CAFL. Overall, CAFL outperforms AFLGo by 3.65x on average. With the data conditions determined by the case C1, CAFL with **T+D** generally outperforms CAFL with **T** and AFLGo, especially in GraphicsMagick 1.4 and jasper 1.900.12 with significant margins. This is because the case C1 leverages the new exception checks in changelogs, which strongly imply the condition of underlying crashes. CAFL still performs better than AFLGo even with weaker cases (C2 and later), but in some cases, the data conditions adversely affect the generation time (yasm 1.3.0).

8 Discussion

Use-cases with manually written constraints. While all the evaluated cases in this paper are based on the auto-generated constraints, there are several promising use-cases of CDGF if developers write manual constraint descriptions. For example, CDGF can be used to help developer's debugging process. If a developer wants to generate an input which follows a specific execution path, the developer can run CDGF with the manual constraints with the representative target sties in a desired order. As another example, developers may utilize program-specific domain knowledge to seek for the root cause of crashes. This can be done by adding suspicious target sites or data conditions to the constraints automatically generated with the crash dumps.

Bugs that require three or more constraints. In some cases, uninitialized values in use-of-uninitialized-value bugs are mediated by a chain of store instructions, where each of stores introduces additional constraint between the two fundamental constraints for creating and using the uninitialized value. Among the bugs we evaluated, one use-of-uninitialized-value bug is mediated by one store instruction, which makes it require three constraints for creating, transferring, and using the uninitialized value (jbig2_mmr.c:88 in jbig2dec v0.16).

Call-stack-overflow bugs may require multiple constraints at the entry of the recurring function to make the execution stack grow deeper. Unfortunately, we could not cover call-stack-overflow bugs in our evaluation, as they often require a high level of grammar fuzzing [19, 20], which CAFL does not support at the moment.

Ineffective scenarios. The current auto-generated data conditions may cause inefficiency if the cause of a crash is completely unrelated to the near-crash conditions. For example, the uninitialized offsets causing some buffer-overflow crashes are irrelevant to the distance derived by the out-of-bound data condition. CAFL currently mitigates the problem with seed scheduling (see §6.3), but how much inefficiency such a data condition can cause needs further investigation.

Program	Bug Location	Bug Type	Case	Data Condition	AFLGo	CAFL		Speedup
						T	T+D	
libming 0.4.8	decompile.c:398	Use after free	C1	`act.p->Constant >poolcounter` [6]	*	64.3 m	45.2 m	-
libsndfile 1.0.28	pcm.c:670	Global buffer overflow	C1	`pchs >0 && pchs != mchs` [5]	*	214.9 m	205.3 m	-
GraphicsMagick 1.4	pict.c:1114	Heap buffer overflow	C1	`row_count+Max*2 >= 0x7fff` [10]	>2000.0 m	>2000.0 m	121.0 m	16.5x
libtiff 4.0.9	pal2rgb.c:196	Heap buffer overflow	C1	`tss_out / tss_in <3` [15]	25.1 m	1.8 m	8.7 m	13.9x
jasper 1.900.12	jas_seq.c:90	Assertion failure	C1	`x &&y >SIZE_MAX / x` [11]	9.4 m	18.4 m	1.0 m	9.4x
jasper 1.900.13‡	jas_seq.c:90	Assertion failure	C1	`xoff >= width || yoff >= height` [3]	10.6 m	1.6 m	2.0 m	6.6x
libsixel 1.8.3	stb_image.h:5052	Assertion failure	C1	`bits <0 || bits >8` [7]	*	24.4 m	21.0 m	-
libtiff 4.0.7	tiff2ps.c:2487	Heap buffer overflow	C2	`cc + nc == tf_bytesperrow` [13]	4.5 m	114.3 m	20.3 m	0.2x
libtiff 4.0.7	tif_unix.c:115	Heap buffer overflow	C2	`s <ns &&row >= imagelength` [2]	32.6 m	11.4 m	17.5 m	2.9x
yasm 1.3.0	intnum.c:415	Use after free	C3	`e->numterm != numterm` [18]	1406.9 m	107.1 m	219.1 m	13.1x
libtiff 4.0.7	tif_fax3.c:413	Heap buffer overflow	C3	`bytes_read != stripsize` [14]	26.4 m	35.5 m	31.2 m	0.8x
libming 0.4.7	parser.c:66	Heap buffer overflow	C4	† [4]	30.1 m	26.5 m	-	1.1x
Average								3.65x

Table 5: PoC generation time comparison. All reproduction times are cut off at 2000 minutes. Underlined times are the shortest. *: AFLGo fails to launch. †: Not applicable. ‡: Incompletely fixed crash at the same location in jasper 1.900.12.

```
CONSTRAINT %free:
  site <free_site>  ; free()

CONSTRAINT %use:
  site <use_site>
  cond "%use.addr == %free.arg0"
```

Figure 11: Hypothetical use-after-free constraint template.

Bugs that require further research. Among the overflow bugs, we have observed that global-buffer-overflow and buffer-underflow bugs are merely benefited from the simple data conditions used now. For global-buffer-overflow bugs, the global buffers mostly serve as look-aside tables rather than regular buffers, eliminating the arithmetic relation between the access offset and the buffer boundary. For buffer-underflow bugs, most programs access the beginning of the buffer so commonly that the data conditions are unable to distinguish inputs. Constructing more sophisticated data conditions for these bugs requires further research.

Issue on distance of pointer conditions. The current definitions of data conditions consider values as arithmetic entities, whose distances can be derived from the arithmetic value differences between them. However, we found there is yet another class of data conditions that are not appropriate to be handled with the arithmetic value differences. The representative case would be data conditions between pointers. The pointer data conditions are problematic because, if two pointers point to different memory objects, their value differences do not carry any semantic meaning.

As an example, Figure 11 shows an hypothetical constraint template for use-after-free bugs. Even if this constraint template has no syntactical problem, the data condition `cond "%use.addr == %free.arg0"` makes little sense, because the smaller integer error between `%use.addr` and `%free.arg0` does not mean they are more likely to be pointing to the same memory object after mutation.

We noted that the similar problem arises in any temporal pointer bugs, such as double-free. To avoid this problem, CAFL currently does not specify such pointer conditions for temporal pointer bugs (nT), and resorts to the "fuzzy" nature of fuzzing to find the crashing input. A reasonable distance metric for pointer conditions still requires further research.

9 Related Work

Directed greybox fuzzing. AFLGo [21] and SemFuzz [44] are the first DGF systems. Published about the same time, they both drive the seed toward a set of target sites in a way to shorten the distance of the seeds to them. Hawkeye [22] improves DGF based on AFLGo by modifying the distance definition to reflect the call trace. However, such conventional DGF systems lack the concept of the ordered target sites and data conditions, which results in the long fuzzing time before exposing targeted crashes. ParmeSan [36] improves distance measuring with dynamic control-flow graphs. Since the distance metric of CDGF does not depend on the type of control-flow graphs, CAFL can also be benefited from dynamic control-flow graph.

Static analysis-assisted directed fuzzing. Some of the directed fuzzing work attempt to leverage static analysis to guide fuzzers toward desired locations. [29, 30] utilizes the crashing execution paths presented by static analyzers rather than the distance metric. [43] performs an online static analysis to determine at which program location a seed becomes unreachable to the target sites. Unfortunately, they either lack the mechanism to facilitate the crash reproduction at the crash location [43] or over-constrain the fuzzing to inaccurately analyzed paths [29, 30].

Targeted analysis with symbolic execution. Compared to fuzzing techniques, symbolic execution techniques have advantages in solving hard branch conditions. As such, hybrid fuzzing techniques [42, 45] utilize the targeted symbolic execution, which specifically solve hard branch conditions where the fuzzer is stuck. Moreover, [26] incorporates the targeted symbolic execution to DGF and *drills through* hard branch conditions where DGF is stuck. In this regard, CAFL can leverage targeted symbolic execution particularly in solving hard branch conditions. It is worth noting that this would require handling well-known issues in performing symbolic execution—e.g., environment modeling such as system/library calls and solving complex symbolic memory references.

ML-based directed fuzzing. NEUZZ [40] incorporates a neural network model that predicts the branch coverage of mutated seeds to increase the branch coverage. FuzzGuard [47] adopts machine learning to improve the effectiveness of DGF by filtering out the mutated inputs if the learned model predicts a given input is unlikely to shorten the seed distance. As stated in [47], FuzzGuard is orthogonal to DGF and can be incorporated to any of targeted fuzzing systems.

Alternative distance metrics. Angora [23] incorporates the distance of integer values to facilitate the branch condition solving. [34] introduces a new distance metric that utilizes the similarity between the call stack of the executed seed and the use-after-free PoC input. Unfortunately, [34] is limited to the vulnerability that belongs to the use-after-free family.

Domain-specific fuzzing. Rather than reaching the targeted locations, some research allow users to manually determine the high-level objective of the fuzzing. FuzzFactory [37] allows a user to write a domain-specific seed creation rule, which in turn creates the more beneficial seeds in favor of the user-custom conditions. However, it lacks the general mechanism to auto-generate the conditions and drive the seeds against them, such as distances.

PoC generation. [31] utilizes symbolic execution to discover a concrete input that reaches the target program location, and [24] generates PoC inputs using symbolic execution. 1dVul [38] utilizes DGF and symbolic execution to generate the 1-day PoC for the patch-released vulnerability. Unlike [24, 31, 38], CAFL does not require symbolic execution.

10 Conclusion

We present CDGF, an augmented DGF that combines the target sites with the data conditions to define constraints, and attempts to satisfy the constraints in the specified order. We define the distance metric for a constraint sequence to prioritize the seeds that better satisfy the constraints, and automatically generate the constraints with seven types of crash dumps and four types of patch changelogs. The evaluation shows the prototype CDGF system CAFL outperforms the representative DGF system AFLGo by 2.88x in 47 real-world crashes, and better performs in PoC generation as the constraints are more explicit.

11 Acknowledgment

This work was supported by SAMSUNG Research, Samsung Electronics Co.,Ltd., and partly supported by National Research Foundation (NRF) of Korea grant funded by the Korean government MSIT (NRF-2019R1C1C1006095). The Institute of Engineering Research (IOER) and Automation and Systems Research Institute (ASRI) at Seoul National University provided research facilities for this work.

References

[1] AFLGo official git repository. https://github.com/aflgo/aflgo.

[2] CVE-2016-10268 patch changelog. https://github.com/vadz/libtiff/commit/5397a417e61258c69209904e652a1f409ec3b9df.

[3] CVE-2016-9390 patch changelog. https://github.com/jasper-software/jasper/commit/ba2b9d000660313af7b692542afbd374c5685865.

[4] CVE-2017-7578 patch changelog. https://github.com/libming/libming/issues/68.

[5] CVE-2017-8365 patch changelog. https://github.com/erikd/libsndfile/commit/fd0484aba8e51d16af1e3a880f9b8b857b385eb3.

[6] CVE-2018-8964 patch changelog. https://github.com/libming/libming/commit/3a000c7b6fe978dd9925266bb6847709e06dbaa3.

[7] CVE-2019-20056 patch changelog. https://github.com/saitoha/libsixel/commit/814f831555ea2492d442e784ab5d594f6a8e2e8d.

[8] Fine free file command. https://www.darwinsys.com/file/.

[9] Git. https://git-scm.com/.

[10] GraphicsMagick 1.4 heap buffer overflow changelog. https://sourceforge.net/p/graphicsmagick/code/ci/8273307fa414bcd00926bf6ae45d11c53b617fe9/.

[11] jasper 1.900.12 assertion-failure patch changelog. https://github.com/jasper-software/jasper/commit/d91198abd00fc435a397fe6bad906a4c1748e9cf.

[12] libFuzzer. https://llvm.org/docs/LibFuzzer.html.

[13] libtiff 4.0.7 tiff2ps.c:2487 patch changelog. https://gitlab.com/libtiff/libtiff/-/commit/ebf0864306f4f24ac25011cf5d752b94c897faa1.

[14] libtiff 4.0.7 tif_fax3.c:413 patch changelog. https://github.com/vadz/libtiff/commit/9657bbe3cdce4aaa90e07d50c1c70ae52da0ba6a.

[15] libtiff 4.0.9 pal2rgb.c:196 heap-buffer-overflow patch changelog. https://gitlab.com/libtiff/libtiff/-/commit/9171da596c88e6a2dadcab4a3a89dddd6e1b4655.

[16] Mercurial SCM. https://www.mercurial-scm.org/.

[17] Whole program LLVM in Go. https://github.com/SRI-CSL/gllvm.

[18] yasm 1.3.0 use-after-free patch changelog. https://github.com/PeterJohnson/yasm/commit/25547a595db288cba1e2aac6e3b1fc3e1c72614e.

[19] Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, Ahmad-Reza Sadeghi, and Daniel Teuchert. NAUTILUS: Fishing for deep bugs with grammars. In *Proceedings of the 26th Network and Distributed System Security Symposium*, 2019.

[20] Tim Blazytko, Cornelius Aschermann, Moritz Schlögel, Ali Abbasi, Sergej Schumilo, Simon Wörner, and Thorsten Holz. GRIMOIRE: Synthesizing structure while fuzzing. In *Proceedings of the 28th USENIX Security Symposium*, 2019.

[21] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. Directed greybox fuzzing. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, 2017.

[22] Hongxu Chen, Yinxing Xue, Yuekang Li, Bihuan Chen, Xiaofei Xie, Xiuheng Wu, and Yang Liu. Hawkeye: Towards a desired directed grey-box fuzzer. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, 2018.

[23] Peng Chen and Chen Hao. Angora: Efficient fuzzing by principled search. In *Proceedings of the 2018 IEEE Symposium on Security and Privacy*, 2018.

[24] Peter Dinges and Gul Agha. Targeted test input generation using symbolic-concrete backward execution. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, 2014.

[25] Brendan Dolan-Gavitt, Patrick Hulin, Engin Kirda, Tim Leek, Andrea Mambretti, Wil Robertson, Frederick Ulrich, and Ryan Whelan. LAVA: Large-scale automated vulnerability addition. In *Proceedings of the 2016 IEEE Symposium on Security and Privacy*, 2016.

[26] Xiaoning Du. Marvel: A generic, scalable and effective vulnerability detection platform. In *Proceedings of the 41st International Conference on Software Engineering: Companion Proceedings*, 2019.

[27] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. Why don't software developers use static analysis tools to find bugs? In *Proceedings of the 35th International Conference on Software Engineering*, 2013.

[28] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*, 2004.

[29] Hongliang Liang, Lin Jiang, Lu Ai, and Jinyi Wei. Sequence directed hybrid fuzzing. In *Proceedings of the IEEE 27th International Conference on Software Analysis, Evolution and Reengineering*, 2020.

[30] Hongliang Liang, Yini Zhang, Yue Yu, Zhuosi Xie, and Lin Jiang. Sequence coverage directed greybox fuzzing. In *Proceedings of the 27th International Conference on Program Comprehension*, 2019.

[31] Kin-Keung Ma, Khoo Yit Phang, Jeffrey S. Foster, and Michael Hicks. Directed symbolic execution. In *Proceedings of the 10th Static Analysis Symposium*, 2003.

[32] Barton P. Miller, Louis Fredriksen, and Bryan So. An empirical study of the reliability of unix utilities. 1990.

[33] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2007.

[34] Manh-Dung Nguyen, S'ebastien Bardin, Richard Bonichon, Roland Groz, and Matthieu Lemerre. Binary-level directed fuzzing for use-after-free vulnerabilities. 2020.

[35] Damien Octeau, Somesh Jha, Matthew Dering, Patrick McDaniel, Alexandre Bartel, Li Li, Jacques Klein, and Yves Le Traon. Combining static analysis with probabilistic models to enable market-scale android inter-component analysis. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2016.

[36] Sebastian Österlund, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. ParmeSan: Sanitizer-guided greybox fuzzing. In *Proceedings of the 29th USENIX Security Symposium*, 2020.

[37] Rohan Padhye, Caroline Lemieux, Koushik Sen, Laurent Simon, and Hayawardh Vijayakumar. FuzzFactory: Domain-specific fuzzing with waypoints. *Proceedings of the ACM on Programming Languages*, 2019.

[38] Jiaqi Peng, Feng Li, Bingchang Liu, Lili Xu, Binghong Liu, Kai Chen, and Wei Huo. 1dVul: Discovering 1-day vulnerabilities through binary patches. In *Proceedings of the 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, 2019.

[39] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. AddressSanitizer: A fast address sanity checker. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, 2012.

[40] Dongdong She, Kexin Pei, Dave Epstein, Junfeng Yang, Baishakhi Ray, and Suman Jana. NEUZZ: Efficient fuzzing with neural program smoothing. In *Proceedings of the 2019 IEEE Symposium on Security and Privacy*, 2019.

[41] Evgeniy Stepanov and Konstantin Serebryany. MemorySanitizer: fast detector of uninitialized memory use in C++. In *Proceedings of the 2015 IEEE/ACM International Symposium on Code Generation and Optimization*, 2015.

[42] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *Proceedings of the 23rd Network and Distributed System Security Symposium*, 2016.

[43] Valentin Wüstholz and Maria Christakis. Targeted greybox fuzzing with static lookahead analysis. In *Proceedings of the 42nd International Conference on Software Engineering*, 2020.

[44] Wei You, Peiyuan Zong, Kai Chen, XiaoFeng Wang, Xiaojing Liao, Pan Bian, and Bin Liang. SemFuzz: Semantics-based automatic generation of proof-of-concept exploits. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, 2017.

[45] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. QSYM : A practical concolic execution engine tailored for hybrid fuzzing. In *Proceedings of the 27th USENIX Security Symposium*, 2018.

[46] Michal Zalewski. American Fuzzy Lop: a security oriented fuzzer. `http://lcamtuf.coredump.cx/afl/`.

[47] Peiyuan Zong, Tao Lv, Dawei Wang, Zizhuang Deng, Ruigang Liang, and Kai Chen. FuzzGuard: Filtering out unreachable inputs in directed grey-box fuzzing through deep learning. In *Proceedings of the 29th USENIX Security Symposium*, 2020.

A Definition of Constraint Language

A.1 Basic Syntax

Figure 12 shows the context-free grammar of the constraint language. A constraint description (*ConstrDesc*) consists of multiple ordered constraints (*Constr*) enumerated in the required satisfaction order. A constraint has a head (*ConstrHead*) that specifies its name and a body (*ConstrBody*) that specifies the target site and the data conditions. A constraint is considered to be satisfied if the target site is reached and all data conditions are satisfied in the order of appearance, if any.

A.2 Constraint Body

A constraint body (*ConstrBody*) specifies a target site (*TargetSite*) and data conditions (*DataCond*). Multiple data conditions need to be satisfied in the specified order. Data conditions are optional, so if no data condition is specified, the constraint is considered to be satisfied when the target site is reached.

A.2.1 Target Site

A target site (*TargetSite*) specifies a targeted source location (*Location*). It includes the file name followed by the column and line numbers (*String*:*Int*:*Int*). When reached, the constraint captures the variables used by the source-level expression pointed by the location. The variables may be captured at a different source location if it is annotated ([*Location*]).

A target site may specify multiple target source locations ((|| *Location*)*). In that case, the target site is considered to be reached if any one of the source locations is reached. The variables are not captured if some source locations have different expression types.

A.2.2 Data Condition

A data condition (*DataCond*) specifies a condition expression (*ConditionExpr*) that must be satisfied at the target site. A data condition can be declared in two types; `cond` and `assert`. If a data condition is declared as `cond`, the distance of the data condition follows the distance derived from the specified condition. If it is declared as `assert`, the distance of the data condition is maxed out unless the condition is satisfied and has zero distance.

A.3 Condition Expressions

A condition expression (*ConditionExpr*) represents a boolean condition in terms of given value expressions. The distance of a condition expression is zero when it is satisfied, otherwise follows the operator-specific distance rules (*CmpOp*). Condition expressions may be combined with logical operators (*LogicOp*), where the distance is calculated based on

$$\begin{aligned}
\mathit{ConstrDesc} &\rightarrow \mathit{Constr}^{*} \\
\mathit{Constr} &\rightarrow \mathit{ConstrHead}\,\texttt{:}\ \mathit{ConstrBody} \\[1ex]
\mathit{ConstrHead} &\rightarrow \texttt{CONSTRAINT}\ \mathit{ConstrNameExpr} \\
\mathit{ConstrNameExpr} &\rightarrow \texttt{\%}\mathit{String} \\[1ex]
\mathit{ConstrBody} &\rightarrow \mathit{TargetSite}\ \mathit{DataCond}^{*} \\[1ex]
\mathit{TargetSite} &\rightarrow \texttt{site}\ \mathit{Location}\ (\|\ \mathit{Location})^{*} \mid \\
&\ \texttt{site}\ \mathit{Location}\ \texttt{[}\mathit{Location}\texttt{]} \\
\mathit{Location} &\rightarrow \mathit{String}\,\texttt{:}\,\mathit{Int}\,\texttt{:}\,\mathit{Int}
\end{aligned}$$

$$\begin{aligned}
\mathit{DataCond} &\rightarrow (\texttt{cond} \mid \texttt{assert})\ \texttt{"}\mathit{ConditionExpr}\texttt{"} \\[1ex]
\mathit{ConditionExpr} &\rightarrow (\mathit{ConditionExpr}) \mid \texttt{none} \mid \\
&\ \mathit{ValueExpr}\ \mathit{CmpOp}\ \mathit{ValueExpr} \mid \\
&\ \mathit{ConditionExpr}\ \mathit{LogicOp}\ \mathit{ConditionExpr} \\[1ex]
\mathit{ValueExpr} &\rightarrow (\mathit{ValueExpr}) \mid \mathit{Int} \mid \mathit{Hex} \mid \mathit{Variable} \mid \\
&\ \mathit{ValueExpr}\ \mathit{ArithOp}\ \mathit{ValueExpr} \\
\mathit{Variable} &\rightarrow \mathit{ConstrNameExpr}\,\texttt{.}\,\mathit{VarIdent} \\
\mathit{VarIdent} &\rightarrow \texttt{value} \mid \texttt{ret} \mid \texttt{size} \mid \texttt{addr} \mid \texttt{readdr} \mid \\
&\ \texttt{endaddr} \mid \texttt{lhs} \mid \texttt{rhs} \mid \texttt{arg}\mathit{Int}
\end{aligned}$$

$$\mathit{Int} \rightarrow \texttt{-}^{?}(\texttt{0-9})^{+}, \qquad \mathit{Hex} \rightarrow \texttt{0x}(\texttt{0-9} \mid \texttt{a-f} \mid \texttt{A-F})^{+}, \qquad \mathit{String} \rightarrow (\texttt{a-z} \mid \texttt{A-Z} \mid \texttt{\_})(\texttt{a-z} \mid \texttt{A-Z} \mid \texttt{0-9} \mid \texttt{\_})^{*}$$

$$\mathit{ArithOp} \rightarrow + \mid - \mid / \mid *, \qquad \mathit{CmpOp} \rightarrow \texttt{==} \mid \texttt{!=} \mid \texttt{<=} \mid \texttt{<} \mid \texttt{>=} \mid \texttt{>}, \qquad \mathit{LogicOp} \rightarrow \texttt{\&\&} \mid \|$$

Figure 12: Context-free grammar of constraint description.

the distances of original expressions. See Appendix B for the distance definitions.

A.4 Value Expressions

A value expression (*ValueExpr*) represents an arithmetic value that can be used with comparison operators. It can be further manipulated by the arithmetic operators (*ArithOp*) when it is evaluated. All values have a canonical 64-bit integer type; pointers are regard as regular 64-bit integers, and floating-point values are casted to fixed-point values before captured.

A.4.1 Variable

A variable (*Variable*) indicates a captured value by the specified constraint. It consists of a constraint name expression that specifies the name of the containing constraint (*ConstrNameExpr*), and the variable identifier that specifies the value to be referenced from the constraint (*VarIdent*).

A variable identifier indicates a variable by its place in the expression where it has been captured (`arg`*Int*). For example, `arg`0 may be the 0th argument of a call expression or the left-hand-side operand of a division expression. Some identifiers are reserved with specific semantic meanings, such as `ret` being the return value if the expression is a call expression. Other reserved identifiers are listed below.

- `value` indicates the resulting value produced by the expression, such as the loaded value if the expression is a dereference expression. This is also an alias of `ret`.
- `addr` indicates the dereferenced address if the expression is a dereference expression.
- `size` and `endaddr` indicate the size and the end address of the allocated memory respectively, if the expression is the recognizable allocation function, such as `malloc`.
- `readdr` indicates the reallocated address if the expression is the recognizable reallocation function, such as `realloc`.
- `lhs` and `rhs` indicate the value of left-hand-side and right-hand-side operands if the expression is a binary expression.

B Distance of Condition Expressions

B.1 Comparison Operators

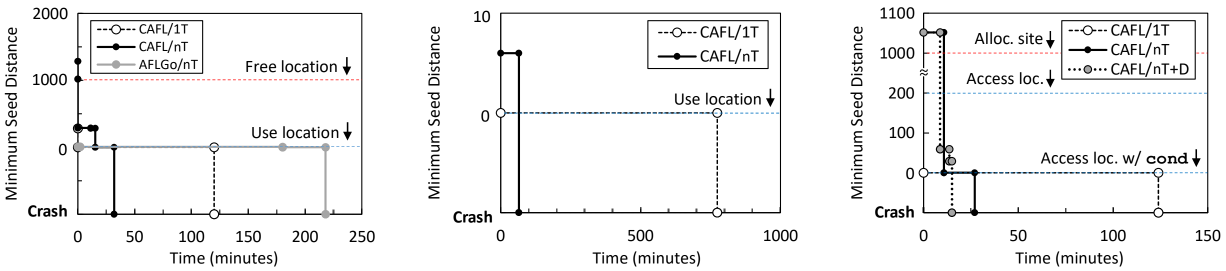
The distance of the condition expression $n_1 \square n_2$, where n_1 and n_2 are integers and $\square$ is a comparsion operator, is calculated based on the absolute difference between n_1 and n_2. Specifically, the distance is the absolue difference if it is not satisfied, or 0 otherwise. The detailed rationales behind the distance of each comparison operator are as follows.

- The distances of ==, >= and <= are equal to the absolute difference of two operands if unsatisfied, or 0 otherwise.
- The distances of > and < are the distances when it is reformulated with >= and <=, respectively. (e.g., $n_1 > n_2 \Rightarrow n_1 >= n_2 + 1$)
- The distance of $!=$ is the distance when it is reformulated with > and <. (i.e., $n_1\ != n_2 \Rightarrow n_1 > n_2 \,\|\, n_1 < n_2$)

The formalized distance of each comparison operator is shown in Table 1.

B.2 Logical Operators

The distance of the condition expression $e_1 \bigcirc e_2$, where e_1 and e_2 are also condition expressions and $\bigcirc$ is a logical operator, is decided by the rules below.



(a) yasm 1.3.0 (Use after free) **(b)** nasm 2.14rc0 (Use after free) **(c)** libming 0.4.7 (Heap buffer overflow)

Figure 13: Minimum observed seed distance change of three representative programs. c_{con} and c_{data} are assumed to 1000 and 100, respectively.

- The distance of && takes the maximum distance of two condition expressions.
- The distance of || takes the minimum distance of two condition expressions.

Formally, if e' is a condition expression where $e' = e_1 \bigcirc e_2$ and $\hat{d}_{\bigcirc}(e')$ is the distance of e', $\hat{d}_{\&\&}(e') = \max(\hat{d}(e_1), \hat{d}(e_2))$ and $\hat{d}_{||}(e') = \min(\hat{d}(e_1), \hat{d}(e_2))$

C Constraint Generation Algorithm for Changelogs

Algorithm 1 shows the algorithm to find a proper constraint with a given patch changelog. Given a set of changed locations, first it classifies each location with pre-defined cases described in §5.2. It then takes a set of locations classified as an earlier case to select target sites and generate data conditions. The algorithm finally ties all selected target sites with a sentinel function to make them into one target site.

Algorithm 1: FindConstraints

```
Input Ls: Set of changed locations. (pre: pre-patch, post:post-patch)
Output Tuple of a target site and data conditions.
foreach L in Ls do
    if IsNewCond(L.post) and BodyContainsExcept(L.post) then
        C1Tars ← L
    else if AreBothConds(L) and AreNotEqualConds(L) then
        C2Tars ← L
    else if OneVarChanged(L) then C3Tars ← L
    else C4Tars ← L
if not Empty(C1Tars) and AllSameCondExpr(C1Tars.post) then
    TSs = GetCondLocs(C1Tars.pre)
    DCs = GetConds(C1Tars.post)
else if not Empty(C2Tars) and AllSameCondExpr(C2Tars.pre) and
  AllSameCondExpr(C2Tars.post) then
    TSs = GetCondLocs(C2Tars.pre), C = GetConds(C2Tars)
    DCs = CreateXorCond(C.pre, C.post)
else if not Empty(C3Tars) and AllSameChangedVars(C3Tars) then
    TSs = GetChangedVarLocs(C3Tars.pre)
    V = GetChangedVar(C3Tars)
    DCs = CreateNECond(V.pre, V.post)
else
    TSs = C4Tars, DCs = []
TS = TieTargetSites(TSs)
return (TS, DCs)
```

D Analysis on Minimum Distance Change

Figure 13 shows the change of the minimum observed seed distance during the fuzzing session of three representative programs in Table 3 and Table 4. To visualize how the minimum observed seed distance changes in various settings, we select one trial and scale the time dimension to the average time of its setting. While the distance change is from one trial, the trends of the distance change were similar in every trial. To describe the distance change more clearly, we convert the original seed distance with $c_{con} = 1000$ and $c_{data} = 100$.

Figure 13a describes the minimum seed distance change of the yasm 1.3.0 use-after-free crash. AFLGo and CAFL with **1T** quickly reach the use location as soon as they begins, but they are stuck at the use location for a long time, as most of the seeds bypass the free location. On the other hands, CAFL with **nT** attempts to reach the use location after reaching the free location, and successfully reaches the use location after 15 minutes. By prioritizing the seeds that reaches both in order, CAFL with **nT** reproduces the use-after-free crash after another 17 minutes. nasm 2.14rc0 in Figure 13b exhibits the similar behavior to yasm 1.3.0, where the crash is reproduced faster by CAFL with **nT** even if CAFL with 1T first reaches the use location.

Figure 13c describes the minimum distance change of CAFL with three different constraint settings in libming 0.4.7. **1T** quickly get zero seed distance by reaching the access location, but it does not reproduce the crash as the seeds may not reach the allocation site, nor satisfy the data conditions. **nT** spends around 10 minutes to find the seeds that pass through both the allocation and access locations, but it waste another 17 minutes at the access location as it does not recognize the data conditions. **nT+D** spends about the same time to reach the allocation site and the access location in order, and it gradually converges to the solution of the data condition. As a result, **nT+D** successfully reproduces the crash only after another 5 minutes.