Constraint-guided Directed Greybox Fuzzing

link: gwangmu-cafl.pdf (lifeasageek.github.io)
proof-of-concept (PoC)

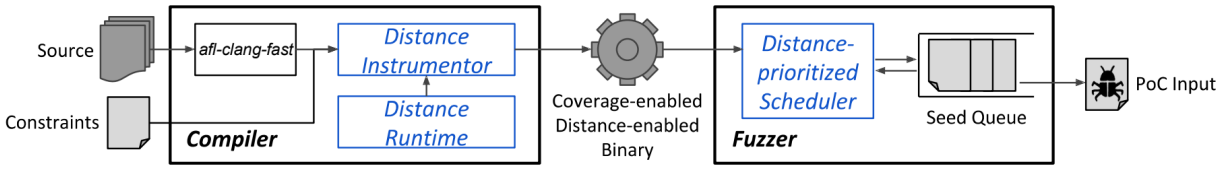

Figure 9: System overview of CAFL.

1 Introduction

Directed grey-box fuzzing (DGF): focus on driving the testing toward a set of specific program locations, called target sites, which allows to intensively fuzz such locations.

DGF techniques take a very long time to identify the targeted crash largely due to the following **two limitations**:

① 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).

② DGF does not consider the data conditions required for the targeted crash and overlook the seeds that satisfy such data conditions. **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 grey-box fuzzing (CDGF)** that resolves the limitations of DGF

--->

① satisfy a sequence of constraints and prioritize the seeds that better satisfy those in order

② present the algorithmic methods to automatically generate the constraints from the additional information sources(crash dumps from memory error detectors & changelogs from patches)

如何测量种子满足约束：定义seed distance based on the distance of the constraints, 而不是定义种子距离 as the average distance to the target sites;

Summary:

1. We present the constraint-guided directed greybox fuzzing (CDGF), which augments the conventional DGF with the ordered target sites and the data conditions.
2. 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.
3. 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.

2 Background

2.1 Directed Grey-box Fuzzing

The directed greybox fuzzing (DGF) 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.

2.2 Usage Example

2.2.1 Static Analyzer Verification

High false alarm rate

2.2.2 Crash Reproduction

Crash reports are often accompanied by a proof-of-concept (PoC) input and a crash dump from memory error detectors.

即使修复了漏洞，也不能保证根源问题被解决(可能只是单纯地让input没有触发crash)。

2.2.3 PoC Generation

利用补丁日志等，去分析补丁在源代码的位置。

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

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

这样会导致follow的程序执行顺序不对。

2.3.2 No Data Condition

DGF cannot drive a seed to such a boundary.

2.4 Requirements

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

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.

CDFG fuzzes in favor of the seeds that are more likely to satisfy all the constraints: **1) if it satisfies more number of constraints and 2) if it is closer to satisfy the first unsatisfied constraint than another**

Distance of an individual constraint as the sum of the distance to the target site and the data conditions;

CDGF combines **the DGF-style distance for target sites** with **the Angora-style distance for data conditions**.

3.2 Example

3.2.1 Ordered Target Sites

完善路径长度计算规则。

FIGURE 2: 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.

3.2.2 Data Conditions

The difference between the size of buf at T1 and the access offset at T2.

4 Constraints

4.1 Definition

A constraint is called satisfied if:

1. the program reaches the target site
2. the data conditions are all satisfied at the target site

Variable capturing

Once the target site is reached, the constraint captures the variables used at the target site.

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.

Orderedness

Constraints may be specified more than one.

4.2 Distance of Constraints

4.2.1 Target Site Distance

Basic block distance

定义了d(B1, B2)是什么

Target site distance

B trace= [B1 ,B2 ,...] ---> DTARGET = d(Bn ,B∗)

4.2.2 Data Condition Distance

Distance of an individual data condition

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

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

where $Var^n(Q)$ is a set of variable vectors captured until the program executes B^n, and $\square$ 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.

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.

4.2.3 Constraint Distance

constraint distance = target site distance + data condition distance.

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_j.

5 Constraint Generation

Design constraint generation for two such sources: **crash dumps from memory error detectors and patch changelogs**.

5.1 Crash Dump

5.1.1 Multiple Target Sites (nT)

The nT template with multiple target sites handles use-after-free, double-free, and use-of-uninitialized-value.

Avoiding wrapper functions

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".

Constraint description

The template specifies multiple target sites that are required to be reached in the specified order to reproduce the crash.

Corresponding bug types

Use-after-free, double-free, and use-of-uninitialized-value correspond to the nT constraint template.

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

2T+D template with two target sites and data conditions handles stack-buffer-overflow and heap-buffer-overflow.

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

1T+D template with one target site and data conditions handles assertion-failure and divide-by-zero.

5.2 Patch Changelog

- `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.

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 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

6.2 CAFL Compiler

Coverage instrumentation:

The CAFL compiler generates the LLVM IR bitcode and annotates the target sites to prevent them optimized out.

Call graph construction:

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:

Inserts the checkpoint calls.

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.

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.

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

1. negative-proportionally to its total distance
2. avoid such local minimum seeds, the CAFL fuzzer exponentially scales down the seed score with respect to the fuzzed times and the stuck depth

Seed creation

1. observe a seed whose score is bigger than the current biggest
2. create seeds in a conventional way

Seed prioritization

The CAFL fuzzer modifies the seed scheduling algorithm of AFL by regulating the selection probability of each seed based on its score.

7 Evaluation

7.1 Microbenchmark: LAVA-1

Present the microbenchmark results using LAVA-1.

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.

7.3 PoC Generation

We measure the PoC generation time with various constraint settings.

8 Discussion

1. Use-cases with manually written constraints
2. Bugs that require three or more constraints

   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.
3. 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.
4. Bugs that require further research

   Among the overflow bugs, we have observed that global-buffer-overflow and bufferunderflow bugs are merely benefited from the simple data conditions used now.
5. Issue on distance of pointer conditions

   There is yet another class of data conditions that are not appropriate to be handled with the arithmetic value differences.

9 Related Work

1. Directed greybox fuzzing
2. Static analysis-assisted directed fuzzing
3. Targeted analysis with symbolic execution
4. ML-based directed fuzzing
5. Alternative distance metric
6. Domain-specific fuzzing

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 a**utomatically 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.