GREYONE: Data Flow Sensitive Fuzzing

Anonymous submission.

Abstract

Data flow analysis (e.g., dynamic taint analysis) has proven to be useful for guiding fuzzers to explore hard-to-reach code and find vulnerabilities. However, traditional taint analysis is labor-intensive, inaccurate and slow, affecting the fuzzing efficiency. Apart from taint, few data flow features are utilized.

In this paper, we proposed a data flow sensitive fuzzing solution GREYONE. We first utilize the classic feature *taint* to guide fuzzing. A lightweight and sound fuzzing-driven taint inference (FTI) is adopted to infer taint of variables, by monitoring their value changes while mutating input bytes during fuzzing. With the taint, we propose a novel input prioritization model to determine which branch to explore, which bytes to mutate and how to mutate. Further, we use another data flow feature *constraint conformance*, i.e., distance of tainted variables to values expected in untouched branches, to tune the evolution direction of fuzzing.

We implemented a prototype of GREYONE and evaluated it on the LAVA data set and 19 real world programs. The results showed that it outperforms various state-of-the-art fuzzers in terms of both code coverage and vulnerability discovery. In the LAVA data set, GREYONE found all listed bugs and 336 more unlisted. In real world programs, GREYONE on average found 2.12X unique program paths and 3.09X unique bugs than state-of-the-art evolutionary fuzzers, including AFL, VUzzer, CollAFL, Angora and Honggfuzz, Moreover, GREYONE on average found 1.2X unique program paths and 1.52X unique bugs than a state-of-the-art symbolic exeuction assisted fuzzer QSYM. In total, it found 105 new security bugs, of which 41 are confirmed by CVE.

1 Introduction

Evolutionary mutation-based fuzzing (e.g., AFL [44]) has become one of the most popular vulnerability discovery solutions, widely used and studied by the community. A core task of such fuzzers is determining the evolution direction, as well as *where* and *how* to mutate seed inputs, in order to efficiently explore hard-to-reach code and satisfy sophisticated data-flow constraints to trigger potential vulnerabilities.

A common solution is utilizing symbolic execution to solve control-flow constraints and help fuzzers to explore code, as proposed in Driller [37], QSYM [43] and DigFuzz [45]. However, symbolic execution is too heavy weight and cannot scale to large applications, and unable to solve many complicated constraints, e.g., one-way functions. Researchers also tried to improve fuzzers with deep learning [29] and reinforcement learning [7], by predicating which byte to mutate and what mutation actions to take. However, they are still in early stage and the improvements are not significant.

Instead, data flow analysis [1](e.g., dynamic taint analysis) has proven to be useful for guiding fuzzing. TaintScope [40] utilized it to locate checksums. VUzzer [30] uses it to identify which bytes and what values are used in branch instructions. Angora [10] uses it to draw the shape of input bytes related to path constraints. These solutions utilize taint to determine where and how to mutate in different ways, and showed good performance in some applications.

1.1 Questions to Address

However, traditional dynamic taint analysis has several limitations. First, it is labor-intensive and requires lots of manual efforts. For example, VUzzer [30] at first only supports x86 platform. In general, these solutions have to interpret each instruction in native or intermediate representation form, with custom taint propagation rules. They also have to build taint models for external function calls or system calls. Second, it is inaccurate. For example, some tainted data values may affect control flow that further affects other data, forming *implicit data flows*. It causes either *under-taint* if the implicit flows are ignored, or *over-taint* if such flows are all counted [19]. Lastly, it is extremely slow (usually several times overheads), making fuzzing inefficient. These seriously limit dynamic taint analysis' application and efficiency in fuzzing. Therefore, the first research question

[1]The paper focuses on fuzzing, and dynamic taint analysis is more accurate than its static counterpart. So we only focus on dynamic taint analysis.

to address is: `RQ1: How to perform lightweight and accurate taint analysis for efficient fuzzing?`

With the inferred taint attributes, VUzzer [30] mutates input bytes used in branch instructions and imprecisely replaces them with expected values (e.g., magic number). REDQUEEN [4] further identifies all *direct copies of inputs*, i.e., input bytes that are directly used in branch constraints (e.g., magic number and checksum), and replaces them with expected values. However, they could neither solve branch constraints related to *indirect copies of inputs*, i.e., input bytes that are transformed and indirectly used in branch constraints, nor prioritize which branch to explore and which bytes to mutate. Thus, the second research question to address is: `RQ2: How to efficiently guide mutation with taint?`

Existing evolutionary fuzzers in general evolve towards increasing code coverage. For example, AFL [44] adds test cases that find new code to the seed queue, and selects one at a time from the queue to mutate. Many other solutions, e.g., AFLfast [6] and CollAFL [14], have been proposed to further improve the way to select seed, accelerating the evolution speed. However, they only considered control flow features rather than data flow features, e.g., taint attributes or constraint conformance, and may waste energies during mutation to explore hard-to-reach branches. Thus, the third research question to address is: `RQ3: How to tune fuzzers' evolution direction with data flow features?`

1.2 Our Solution

We proposed a novel data flow sensitive fuzzing solution GREYONE, to address the aforementioned questions.

Fuzzing-driven Taint Inference (FTI). We first propose FTI to infer taint of variables by conducting a *pilot fuzzing* phase, during which we systematically mutate each input byte (one at a time) and monitor variables' values. If a variable's value changes while an input byte is mutated, we could infer the former is tainted and depends on the latter.

This inference is sound, i.e., without over-taint issues. It is also immune to under-taint issues caused by implicit flows or external calls[2]. Experiments showed that, FTI is more accurate than traditional taint analysis, e.g., able to find 2 to 4 times more dependencies (with no false positives). Furthermore, it avoids the labor-intensive efforts of composing taint propagation rules and is very fast at runtime. *This lightweight and sound solution could scale to large programs, and provide supports for other application scenarios beyond fuzzing.*

Taint-Guided Mutation. Input bytes contribute differently to code coverage. We utilize taint provided by FTI to sort input bytes. More specifically, we prioritize input bytes that affect more untouched branches to mutate, and prioritize untouched branches that depend on more prioritized input bytes to explore. When exploring a branch, we mutate its dependent input bytes in priority order, by precisely replacing direct copies of inputs with expected values (and minor variations).

Conformance-Guided Evolution. Lots of fuzzers (e.g., AFL) use *control flow features*, e.g., code coverage, to guide evolution. To efficiently explore hard-to-reach branches (e.g., those related to indirect copies of inputs), we propose to use complementary *data flow features* to tune the evolution direction. Note that, for each tainted variable used in untouched branches, we need to flip some bits to match the expected values. The amount of efforts required is related to the *constraint conformance*, i.e., the distance of tainted variables to the values expected in untouched branches.

We use this data flow feature to tune the fuzzer's evolution direction. First, we add test cases with higher conformance to the seed queue, making the fuzzer gradually improve the overall conformance and eventually satisfy the constraints of untouched branches. Then, we prioritize seeds with higher conformance to be selected from the queue for mutation, accelerating the exploration of new branches. This evolution could satisfy the constraints in a faster pace, like the gradient descent used in Angora [10]. But it could avoid getting stuck in local minimum and brings long-term stable improvements. Furthermore, we rebase ongoing mutations onto new seeds with higher conformance on-the-fly. Experiments showed that it thus significantly improves the mutation efficiency.

1.3 Results

We implemented a prototype of GREYONE and evaluated it on the LAVA-M dataset [12] and 19 open source applications.

Our taint analysis engine FTI outperforms the classic taint analysis solution DFSan [2]. On average, it finds 1.3X more untouched branches that are tainted (i.e., depending on input bytes), and generates 1X more unique paths during fuzzing.

GREYONE outperforms 6 state-of-the-art evolutionary fuzzers, including AFL and CollAFL [14], in terms of both code coverage and vulnerability discovery. In the LAVA data set, GREYONE finds all listed bugs and 336 more unlisted bugs. In real world applications, GREYONE finds 2.12X unique paths, 1.53X new edges, 6X unique crashes and 3.09X bugs, than the second best counterpart.

In addition, GREYONE demonstrates very good performance in bypassing complicated program constraints, even better than the state-of-the-art symbolic execution assisted fuzzer QSYM [43]. In the real world applications, GREYONE finds 1.2X unique paths, 1.12X new edges, 2.15X unique crashes and 1.52X bugs than QSYM.

In total, GREYONE has found 105 unknown vulnerabilities in these applications. After reporting to upstream vendors, we learned that 25 of them are known by vendors (but not public). Among the remaining 80 bugs, 41 are confirmed by CVE.

To summarize, we make the following contributions:

-

[2] FTI could suffer from under-taint issues due to incomplete pilot fuzzing.

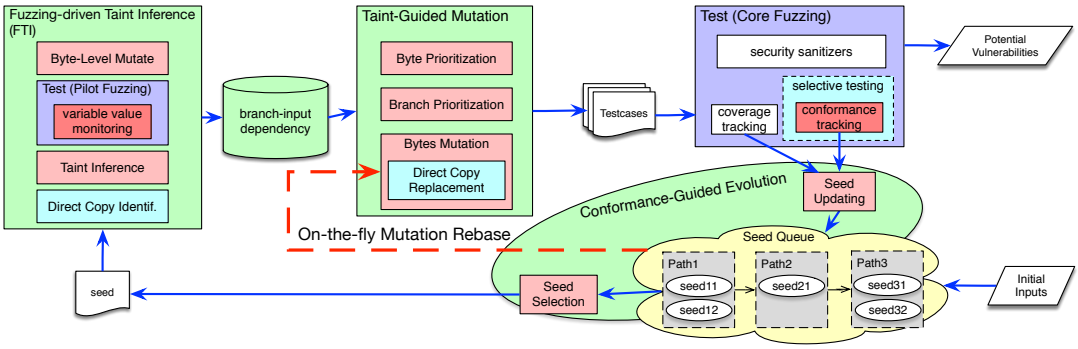

Figure 1: Architecture of GREYONE.

- We propose a taint-guided mutation strategy, able to prioritize which branch to explore and which input bytes to mutate, and determine how to (precisely) mutate.
- We propose a new conformance-guided evolution solution to tune the direction of fuzzing, by taking into consideration data flow features including taint attributes and constraint conformance.
- We implement a prototype GREYONE, evaluate it on 19 widely-tested open source applications, showing that it outperforms various state-of-the-art fuzzers.
- We find 105 unknown vulnerabilities in 19 applications, and help the vendors improve the products.

2 Design of GREYONE

As shown in Figure 1, the overall workflow of GREYONE is similar to AFL, consisting of steps like seed generation/updating, seed selection, seed mutation and testing/tracking.

First, we introduce a new step into the fuzzing loop, i.e., fuzzing-driven taint inference (FTI), to infer taint of variables. We conduct a pilot fuzzing by performing byte-level mutation on the input seed and testing them. During the pilot fuzzing, we monitor program variables' value changes. Once a variable's value changes, we could induce that it is tainted and depends on the mutated input bytes. Besides, we could also identify all tainted variables that use direct copy of inputs.

Second, with the taint attributes provided by FTI, we further guide the fuzzer to mutate seeds in a more efficient way. We prioritize which input bytes to mutate and which branch to explore. In addition, we determine how to mutate input bytes, including direct and indirect copies of inputs.

Lastly, we tune the fuzzing direction with conformance-guided evolution. In addition to code coverage, we track tainted variables' constraint conformance during testing, and add test cases with higher conformance to the seed queue, making the fuzzer gradually increase the conformance and reach untouched branches. Then, we prioritize seeds with higher conformance to select from the queue, accelerating the evolution. Furthermore, once we find a new seed with higher conformance, we rebase ongoing mutations onto this new seed on-the-fly, promoting the mutation efficiency.

2.1 Fuzzing-driven Taint Inference

As shown in [10, 30], taint analysis could guide fuzzers towards efficient mutation and help explore hard-to-reach branches. However, traditional solutions are labor-intensive, slow and inaccurate. GREYONE introduces a lightweight and sound solution, i.e., fuzzing-driven taint inference (FTI).

Intuition. If a variable's value changes after we mutate one input byte, we could infer that the former depends on the latter, either explicitly or implicitly. Furthermore, mutating this input byte could change the constraints of branches that use this variable, leading to new branch exploration.

Interference Rule for FTI. Assume we have a program variable *var* (at a given line of instruction) and a seed input S, and another input $S[i]$ which is derived by mutating the i-th byte of the input S, let $v(var, S)$ be the value of *var* when given the input S. We claim the variable *var* depends on the i-th byte of input S, if the following condition holds.

$$v(var, S) \neq v(var, S[i]) \quad (1)$$

Moreover, if either operand variable of a branch instruction *br* depends on the i-th byte of input S, we claim this branch *br* depends on this input byte. In other words, if the data flow from the input byte to the branch does not satisfy the non-intererence rule [16], the latter depends on the former.

Unlike traditional instruction-level taint analysis, e.g., TaintInduce [46], this rule captures high-level dependency and is more accurate. As discussed later, it has fewer false positives (i.e., over-taint) and false negatives (i.e., under-taint).

2.1.1 Taint Inference

Following the aforementioned intuition and interference rule, FTI infers the taint attributes in a pilot fuzzing phase, which could be integrated with the deterministic fuzzing stage of AFL, with the following three steps, as shown in Algorithm 1.

Byte-Level Mutation. We mutate the seed inputs one byte at a time, with a set of predefined mutation rules (e.g., single-bit flipping, multiple-bits flipping and arithmetic operations). For each seed input `S` and each input offset `pos`, a set of new test cases `BLM(S, pos)` could be derived in this way.

Algorithm 1 Fuzzing-driven Taint Inference.

Input: $seed$
Output: $\{br.taint[seed] \mid br \in branches(\mathbb{P})\}$

```
// Target program is instrumented to collect information, as ℙ'
State = Execute(ℙ', seed)
for each candidate mutation method Opr do
    for each available mutation operand Opd do
        for each position pos in the seed do
            seed' = Mutate(seed, Opr, Opd, pos)
            State' = Execute(ℙ', seed')
            for br ∈ uncovered_branches(State) do
                for var ∈ br do
                    if State(var) ≠ State'(var) then
                        br.taint[seed] ∪= {pos}
                    end if
                end for
            end for
        end for
    end for
end for
```

Note that, we do not mutate multiple bytes at the same time,[3] due to the following reasons. First, we cannot precisely determine which byte is responsible for the potential value change if multiple bytes are mutated, causing either under-taint or over-taint issues. Second, single-byte mutation yields fewer test cases and introduces fewer performance overheads.

Variable Value Monitoring. We then feed the generated test cases to test, and monitor program variables' values during testing. To support the monitoring, we instrument the target applications with special value tracking code.

Note that, we could monitor all program variables in this way. However, for the purpose of fuzzing, we only monitor variables that are used in path constraints. First, it is much faster to monitor fewer variables. Second, only these variables will affect the path exploration, and it is sufficient to only monitor them in order to explore all branches.

Taint Inference. Lastly, after testing each set of test cases `BLM(S, pos)`, we check whether the value of each variable used in path constraints keeps intact or not. If the value of a variable `var` changes, we could infer that `var` is tainted and depends on the `pos`-th byte of the input seed `S`.

As shown in Listing 1, which is extended from REDQUEEN [4], we could detect the value of variable `var1` used in the branch at Line 20 changes, when we mutate either the 20th, 21st, 22nd or 23rd byte of the input. Therefore, `var1` depends on these four bytes.

2.1.2 Comparison with Traditional Taint Analysis.

Comparing to traditional taint analysis, FTI requires fewer manual efforts, and is much more lightweight and accurate.

Manual Efforts. Traditional taint analysis (e.g., [20]) requires labor-intensive efforts. In general, each instruction/statement has to be either interpreted with custom instruction-specific taint propagation rules, or lifted/translated to an intermediate representation form and then analyzed with general taint propagation rules. FTI is architecture independent and requires no extra efforts to port to new platforms.

[3] This may cause incomplete testing.

Listing 1: Motivating example of FTI

```
1  // magic number: direct copy of input[0:8] vs. constant
2  if(u64(input) == u64("MAGICHDR")){
3      bug1();
4  }
5  // checksum: direct copy input[8:16] vs. computed val
6  if(u64(input+8) == sum(input+16, len-16)){
7      bug2();
8  }
9  // length: direct copy of input[16:18] vs. constant
10 if( u16(input+16) > len )) { bug3(); }
11 // indirect copy of input[18:20]
12 if(foo(u16(input+18))==...){ bug4(); }
13 // implicit dependency: var1 depends on input[20:24]
14 if(u32(input+20) == ...){
15     var1 = ...;
16 }
17 // var1 may change if input[20:24] changes
18 // FTI infers: var1 depends on input[20:24]
19 if(var1 == ...){     bug5(); }
```

Speed. FTI is very fast. First, it is based on static code instrumentation, rather than dynamic binary instrumentation. Second, it only monitors values of variables used in path constraints, not all program variables. Third, it does not need to interpret individual instructions with custom rules.

Accuracy. FTI is more accurate than traditional taint analysis solutions. Its inference rule is sound. If a variable is reported to depend on a specific input byte, then it is most likely to be true. In other words, it has no over-taint issues.

It also has fewer under-taint issues. In practice, most under-taint issues are caused by ubiquitous implicit data flows and loss in external functions or system calls. FTI is immune to these cases. However, FTI may still have under-taint issues due to incomplete fuzzing caused by byte-level mutation.

Figure 2 demonstrated how FTI works. Unlike traditional dynamic taint analysis, which focuses on instructions and suffers from over-taint and under-taint issues, FTI could improve the accuracy with fewer efforts.

Head-to-Head Comparison. Note that, several recent works have similar ideas or comparable results. TaintInduce [46]could infer taint propagation rules for each instruction without manual efforts. But it is extremely slow in taint rule inferring stage, due to its mutation on each instruction. ProFuzzer [42] mutates one input byte at a time too. But it monitors the coverage changes rather than value changes, unable to infer taint dependency. MutaFlow [26] monitors changes in sink APIs and could tell whether a parameter is tainted. But it focuses on APIs rather than variables, and cannot provide precise taint information for variables. Furthermore, it lacks a systematic testing, such as the pilot fuzzing performed by FTI, and thus has much more under-taint issues.

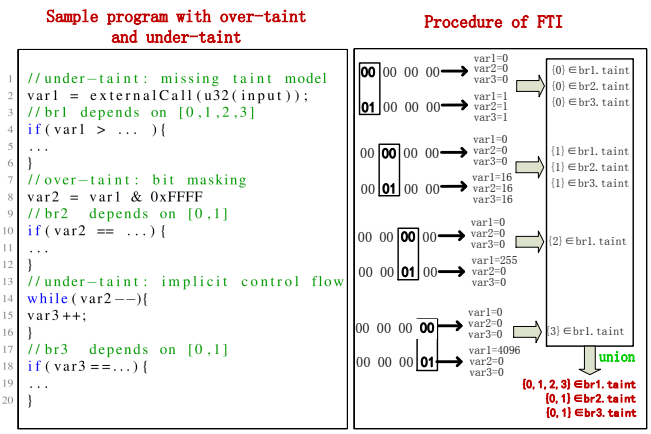

Figure 2: Illustration of the procedure for FTI, along with a sample program with over-taint and under-taint issues.

2.1.3 Identify Direct Copies of Inputs.

It is common that, some input bytes will be directly copied to variables, and compared against expected constants or computed values in branch instructions, as shown at Line 2 (magic number), Line 6 (checksum) and Line 10 (length check) in Listing 1. These input bytes should be replaced with the exact values (or with minor variations like ± 1) expected in the branches, to bypass the hard-to-reach path constraints.

FTI could identify all direct copies of inputs in an efficient way. For each tainted variable used in branch instructions, we could match it against its dependent input bytes. If their values are equal, we report the variable as a direct copy of input. Otherwise, we report it as an indirect copy of input.

2.2 Taint-Guided Mutation

Mutation-based fuzzers will mutate seed inputs in certain ways and generate new test cases, to explore new code and trigger potential vulnerabilities. GREYONE utilizes taint provided by FTI to prioritize which bytes to mutate and which branch to explore, as well as determine how to mutate.

2.2.1 Prioritize Bytes to Mutate

As pointed by [29], not all inputs bytes are equal. Some bytes should be prioritized to mutate, to get a better fuzzing yields. We argue that, *if an input byte could affect more untouched branches, then it should be prioritized over other input bytes*, because mutating this input byte is more likely to trigger untouched branches, and trigger more complicated program behaviors since more branch states have changed.

As shown in Figure 3, each input byte at offset `pos` of a seed input `S` may affect multiple variables, and then affect multiple branches among which some are not explored by any test case. We define a byte's weight as the count of untouched branches depending on this byte, as follows.

$$W_{byte}(S, pos) = \sum_{br \in Path(S)} IsUntouched(br) * DepOn(br, pos) \quad (2)$$

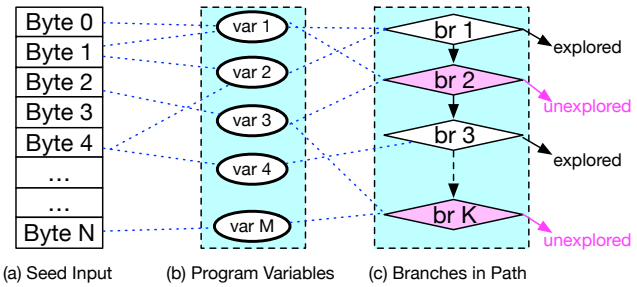

Figure 3: Dependency between inputs, variables and branches.

where, `IsUntouched` returns 1 if the branch `br` is not explored by any test case so far, otherwise 0. And the function `DepOn` returns 1 if the branch `br` depends on the `pos`-th input byte, according to FTI, otherwise 0.

2.2.2 Prioritize Branches to Explore

As shown in Figure 3, a program path may have multiple untouched neighbour branches. Similarly, some untouched branches should be prioritized to explore in order to get a better fuzzing yields. We argue that, *an untouched branch that depends on more high-weight input bytes should be prioritized over other untouched branches*.

If an untouched branch depends on more high-weight input bytes, to explore this branch, we will mutate its dependent input bytes. As aforementioned, mutating these high-weight input bytes is more likely to trigger untouched branches (including branches different from the one to explore).

Accordingly, for a seed `S`, we evaluate the weight of an untouched branch `br` in the according path as the sum of all its dependent input bytes' weight, as follows.

$$W_{br}(S, br) = \sum_{pos \in S} DepOn(br, pos) * W_{byte}(S, pos) \quad (3)$$

2.2.3 Determine Where and How to Mutate

With the weight of input bytes and unexplored branches, we could further determine the seed mutation policy.

Where to mutate? Given a seed and the program path it exercises, we will explore the untouched neighbor branches along this path one by one, in descending order of branch weight according to Equation 3.

When exploring a specific untouched neighbor branch, we will mutate its dependent input bytes one by one, in descending order of byte weight according to Equation 2.

How to mutate direct copies of input? As aforementioned, direct copies of inputs should match the values expected in untouched branches. Thus, during mutation we replace the direct copy of input bytes with the exact expected values (for magic number and checksum etc.) and values with minor variations (e.g., ± 1 for length checks etc.).

The core question left is how to get the expected values. There are two cases. If a constant value (e.g., magic number) is

expected, we record this constant value with FTI. If a runtime-computed value (e.g., checksum) is expected, we first feed a malformed input to test, and get the expected runtime value with FTI. Then we use the recorded value (and with minor variations) to patch the dependent input bytes.

Note that, REDQUEEN [4] could also mutate direct copies of input bytes. Unlike GREYONE, REDQUEEN could not precisely locate the exact position of dependent bytes. It has to mutate the seed hundreds of times to get a colorized version with higher entropy, which exercises the same path. The colorized version is tested again, and compared with the original seed, to locate the potential positions of dependent bytes. The colorization process is very slow, and the number of candidate positions could be large too. As a result, it wastes more time to precisely mutate the dependent bytes than GREYONE.

How to mutate indirect copies of input? If some input bytes affect an untouched branch but their direct copy is not used in the branch, we will mutate these bytes one by one, in descending order of byte weight according to Equation 2.

More specifically, we will apply random bit flipping and arithmetic operations on each dependent byte. Different from the byte-level mutation used in FTI, multiple dependent bytes could be mutated together in this phase.

As discussed later, our conformance-guided evolution solution will rebase the mutation onto better seeds on-the-fly, which could greatly improves the mutation of indirect copies.

Mitigate the under-taint issue. As aforementioned, FTI may have under-taint issues due to incomplete testing. Thus, for any untouched branch, its dependent input bytes reported by FTI could be incomplete. In order to explore that branch, we have to mutate the missing dependent input bytes as well.

More specifically, in addition to mutate the dependent input bytes reported by FTI, we also randomly mutate their adjacent bytes with a small probability.

2.3 Conformance-Guided Evolution

A wide range of fuzzers (e.g., AFL) use *control flow features*, e.g., code coverage, to guide evolution direction of fuzzing. To efficiently explore hard-to-reach branches (e.g., those related to indirect copies of inputs), we propose to use complementary *data flow features* to tune the evolution direction of fuzzing.

We note that, for each tainted variable used in untouched branches, we need to flip some bits of its dependent input bytes to make it match the expected value. Some test cases require fewer efforts (i.e., bit flipping) than others. The amount of efforts required is related to the constraint conformance, i.e., the distance of tainted variables to the values expected in untouched branches. Seeds with higher conformance are more likely to yield test cases exercising untouched branches.

Based on this observation, we use the seed's constraint conformance to tune the evolution direction of fuzzing. We modify the seed updating and seed selection policies accordingly, to drive the fuzzer towards this direction. The test cases generated during fuzzing are more likely to have higher conformance and eventually satisfy the hard-to-reach constraints.

2.3.1 Conformance Calculation

The constraint conformance indicates how much the target (e.g., seed) matches with the path constraints.

Conformance of an untouched branch. Given an untouched branch `br`, which relies on two operands `var1` and `var2`, we define its constraint conformance as follows.

$$C_{br}(br,S) = NumEqualBits(var1, var2) \tag{4}$$

where, the function `NumEqualBits` returns the number of equal bits between the two arguments. Note that, for a branch in a switch statement, the two variables it relies on are the switch condition and the case value.

Conformance of a basic block. Given a seed `S` and a basic block `bb` it has explored, `bb` may have multiple untouched neighbor branches (e.g., switch statements). We define the constraint conformance of `bb` as the maximum conformance of all its untouched neighbor branches:

$$C_{BB}(bb,S) = \underset{br \in Edges(bb)}{MAX} IsUntouched(br) * C_{br}(br,S) \tag{5}$$

Conformance of a test case. Given a test case `S`, its constraint conformance is defined as the sum of the conformance score of all basic blocks it has explored.

$$C_{seed}(S) = \sum_{bb \in Path(S)} C_{BB}(bb,S) \tag{6}$$

Note that, seeds with higher constraint conformance are likely to have (1) more untouched neighbor branches, and (2) individual untouched branches with higher constraint conformance. Further mutations could thus quickly trigger more untouched branches or target individual untouched branches.

2.3.2 Conformance-Guided Seed Updating

In addition to test cases that find new code, we also add test cases with higher constraint conformance to the seed queue. In order to efficiently support this new seed updating scheme, we proposed a novel seed queue structure.

Two-Dimensional Seed Queue. Traditional seed queues are usually kept in a linked list, where each node represents a seed that explores a unique path[4]. We extend each node to include multiple seeds that explore the same path and have the same conformance but different block conformance, to form a two-dimensional seed queue, as shown in Figure 4.

Seed queue Updates. Figure 4 also shows how we update the seed queue, in the following three cases.

- **A. New path.** If the test case finds new code, then it will be added to the seed queue as a new node, same as other coverage-guided fuzzers (e.g., AFL).

[4] In AFL, it represents a unique edge hit or a new edge hit count range

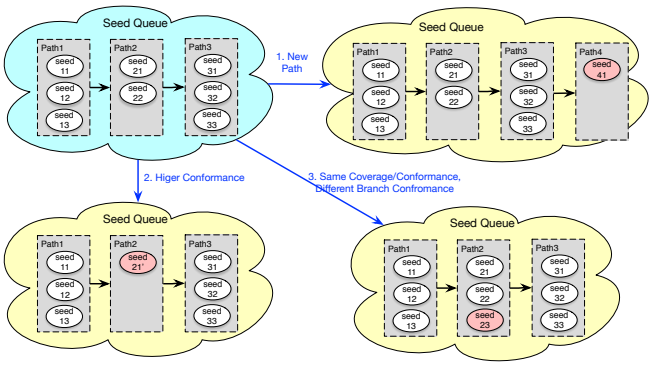

Figure 4: Dynamics of seed queue updating.

- **B. Same path but higher conformance.** If the test case does not find any new code, but has a higher conformance than seeds in the corresponding node (with same path) in the queue, then this node will be replaced with a new node consisting of only this test case.
- **C. Same path and conformance, but different basic block conformance.** If the test case explores the same path and has the same conformance as seeds in the corresponding node in the queue, but has a distribution of basic block conformance different from seeds in that node, then we will append this test case to that node.

It is worth noting that, in the last case, since the test case has a unique distribution of basic block conformance, it could derive new test cases to quickly trigger untouched neighbor branches of some basic blocks, and thus is useful.

Comparison. This seed updating policy makes the fuzzer gradually improve the overall conformance, and satisfies the constraints of untouched branches with a fast pace, at a speed comparable to the gradient descent algorithm used in Angora [10]. But it could avoid getting stuck in local minimum like Angora, and brings long-term stable improvements.

Note that, honggfuzz [38] also compares the equality of operands in branch statements. If a branch's equality increases, it adds the test case to the seed queue. However, it does not exclude compare instructions related to touched branches, which are useless to branch exploration. Further, a basic block may have multiple compare instructions inside, but not all of them are related to branches. Lastly, it lacks the efficient two-dimensional seed queue structure proposed in this paper, limiting its efficiency as well.

2.3.3 On-the-fly Mutation Rebase

Once we find a test case exercising the same program path as previous seeds but has a higher conformance, i.e., case B as aforementioned, we not only add this test case to the seed queue by replacing the corresponding node with a new node, but also replace all uses to the seeds being replaced.

Especially, if the seed being replaced is used by an ongoing mutation, we will rebase the mutation onto the new seed, since the new seed is better. This operation could be done on-the-fly, as illustrated in red line in Figure 1. Experiments showed that, this optimization technique is very effective. For example, it promotes the speed of finding the same number of bugs in the LAVA data set by three times.

2.3.4 Conformance-Guided Seed Selection

Many works [6, 14] have proved that seed selection policies could accelerate the evolution of fuzzing. We propose to prioritize seeds with higher conformance during seed selection.

More specifically, we iterate the linked list of the seed queue, and select linked nodes that have higher conformance with a higher probability. Then a random seed in this linked node will be selected for further mutation.

With this scheme, seeds with higher conformance are more likely to be selected. Further mutations are more likely to yield test cases with higher conformance, which could satisfy the hard-to-reach constraints of untouched branches.

3 Implementation

We implemented a prototype of GREYONE with over 20,000 lines of C/C++ code. The current prototype supports analyzing applications with LLVM bytecode. Here, we present some of its implementation details.

3.1 Modularized Framework

As shown in Figure 1, GREYONE consists of several core components, e.g., seed updating, seed selection, seed mutation and testing. We implemented a set of extensible interfaces to support various policies and future improvements.

Test Case Scoring. Evolutionary fuzzers usually put some test cases to a seed pool for further mutation according to a certain test case scoring algorithm. We implemented a general interface of test case scoring, able to support both the coverage-guided seed updating policy adopted by AFL and the conformance-guided policy adopted by GREYONE.

Seed Prioritization. Fuzzers usually prioritize seeds to select and assign different energy to mutate according to a certain seed scoring algorithm. We implemented a general interface of seed scoring, able to support the conformance-guided seed selection policy adopted by GREYONE and policies used by other fuzzers (e.g., CollAFL [14] and AFLfast [6]).

Seed Mutation Algorithms. In addition to the the mutation operators (e.g., byte flipping) implemented by other fuzzers (e.g., AFL), we also add supports to byte-level mutation used by FTI, and direct-copy mutation in which the fuzzer is told the exact offset and exact value to use.

State Manager. The fuzzer usually requires special data structures to support efficient communication between components and efficient decision making. We constructed many

tree-based and hash-table-based structures to store these information, including control flow graph, code coverage, seed conformance, variables' taint attributes and variables' values.

Selective Testing. In addition to code coverage tracking, GREYONE has two more modes during testing: (1) variable value monitoring mode used for FTI; (2) conformance-guided tracking mode for evolution tuning. To efficiently schedule these different testing modes, we extend the fork server used by AFL to switch between them on demand. For example, during fuzzing, if a seed has spent too much mutation energy or the conformance does not increase for a while, then we will switch from conformance tracking mode to regular coverage tracking mode.

3.2 Static Analysis and Instrumentation.

To support the policies proposed in the paper, we need to first analyze the target applications with static analysis, as well as collect some information at runtime.

We perform some basic inter-procedural control flow analysis with the help of Clang, and get the control flow graph and other necessary information.

Coverage Tracking. As pointed by CollAFL [14], there is a serious hash collision issue in traditional coverage tracking solutions (e.g., AFL). We reproduce the mitigation solution of CollAFL in GREYONE.

Conformance Tracking. To support conformance tracking, we instrument each branch statement (including conditional branches and switch statements) to count the number of equal bits of its operands (by operations like `__builtin_popcount`).

Variable Value Monitoring FTI relies on variable value monitoring during fuzzing. We instrument the application to record the values of variables used in path constraints. More specifically, we assign unique IDs to all such variables, and store their values in a bitmap (with the ID as key), similar to the bitmap storing code coverage used by AFL.

4 Evaluation

In this section, we evaluated the efficiency of GREYONE, and showed its improvements compared to other fuzzers.

4.1 Experiment Setup

Following the guidance in [21], we conducted the experiments carefully, to draw conclusions as objective as possible.

Baseline fuzzers to compare. We compared GREYONE against several well-known evolutionary mutation-based fuzzers, including AFL [44], VUzzer [30], Angora [10], CollAFL [14] , Honggfuzz [38], and QSYM[43][5]. They are chosen based on the following considerations. First, AFL was the most popular baseline fuzzer studied in the community. Second, Angora and VUzzer also utilized taint to guide fuzzing. Third, CollAFL provides more accurate coverage information, which is also utilized by GREYONE. In addition, CollAFL proposed a seed selection policy relying on control flow features, different from GREYONE. Further, Honggfuzz is a core fuzzing engine in Google's OSS-Fuzz platform [33], and also uses light-weight data tracking to identify good seeds. Lastly, QSYM is a popular symbolic execution assisted fuzzer, and we can use it to evaluate GREYONE' capability on bypassing complicated program constraints.

Target applications to test. We chose target applications considering several factors, including popularity, frequency of being tested, development activeness, and functionality diversity. Finally, we chose 19 popular open source Linux applications (in latest version when tested), including well-known development tools (e.g., `readelf`, `nm`, `c++filt`), image processing libraries (e.g., `libtiff`), document processing libraries (e.g., `libwpd`), terminal processing libraries (e.g., `libncurses`), audio or video processing libraries (e.g., `ibsndfile`), code processing tools (e.g., `cflow`, `bison`, `nasm`), graphics processing libraries (e.g., `libcaca` and `libsixel`), and data processing libraries (e.g., `libsass` and `libxsmm`) etc. Furthermore, we also evaluated GREYONE on the LAVA-M data set [12] as other fuzzers.

Performance metrics. We chose vulnerability discovery and code coverage as two major metrics used to compare the efficiency of each fuzzer with GREYONE. For code coverage, we mainly considered path coverage (i.e., number of seeds in the queue) and edge coverage (i.e., number of edge hit) similar to [14, 42]. For vulnerability discovery, we tracked the growth trend of unique crashes detected by different fuzzers. We further utilized tools including afl-collect [3], AddressSanitizer [34] and UBSan [23] to deduplicate redundant crashes and identify unique vulnerabilities.

Note that, fuzzers have different representations of fuzzing states (e.g., bitmap). We therefore slightly modify them to get unified fuzzing states and perform fair comparison.

Initial seeds. Note that, our taint analysis engine FTI relies on byte-level mutation. It will perform poorly if no initial seeds are given, lowering the efficiency of GREYONE. Therefore, we did not test target applications with empty seeds. Instead, we test each target application with 10 initial seeds.

For each target application, we randomly downloaded about 100 input files from the Internet, according the required input file formats. Then, we use the tool afl-cmin shipping with AFL [44], to filter out a minimal subset of inputs that have the same code coverage. Finally, we randomly selected 10 inputs from these distilled inputs, and used them as the initial seeds.

Randomness mitigation. Since mutation-based fuzzers all rely on random mutation, there could be performance jitter during testing. We took two actions to mitigate the randomness issue. First, we perform each experiment for 5 times, and evaluate the average performance as well as the minimal

[5] CollAFL is not open source. We implemented a copy following its design. Another work REDQUEEN [4] is also related, but it is disclosed only one month ago and not open source. Thus we are unable to compare with it.

Table 1: Number of vulnerabilities (accumulated in 5 runs) detected by 6 fuzzers, including AFL, CollAFL-br, VUzzer, Honggfuzz, Angora, and GREYONE, after testing each application for 60 hours.

Applications	Version	AFL	CollAFL- br	Honggfuzz	VUzzer	Angora	GREYONE	Vulnerabilities by GREYONE		
								Unknown	Known	CVE
readelf	2.31	1	1	0	0	3	4	2	2	-
nm	2.31	0	0	0	0	0	2	1	1	*
c++filt	2.31	1	1	1	0	0	4	2	2	*
tiff2pdf	v4.0.9	0	0	0	0	0	2	1	1	0
tiffset	v4.0.9	1	2	0	0	0	2	1	1	1
fig2dev	3.2.7a	1	3	2	0	0	10	8	2	0
libwpd	0.1	0	1	0	0	0	2	2	0	2
ncurses	6.1	1	1	0	0	0	4	2	2	2
nasm	2.14rc15	1	2	2	1	2	12	11	1	8
bison	3.05	0	0	1	0	2	4	2	2	0
cflow	1.5	2	3	1	0	0	8	4	4	0
libsass	3.5-stable	0	0	0	0	0	3	2	1	2
libbson	1.8.0	1	1	1	0	0	2	1	1	1
libsndfile	1.0.28	1	2	2	1	0	2	2	0	1
libconfuse	3.2.2	1	2	0	0	0	3	2	1	1
libwebm	1.0.0.27	1	1	0	0	0	1	1	0	1
libsolv	2.4	0	0	3	2	2	3	3	0	3
libcaca	0.99beta19	2	4	1	0	0	10	8	2	6
liblas	2.4	1	2	0	0	0	6	6	0	4
libslax	20180901	3	5	0	0	0	10	9	1	*
libsixl	v1.8.2	2	2	2	2	3	6	6	0	6
libxsmm	release-1.10	1	1	2	0	0	5	4	1	3
Total	-	21	34	18	6	12	105 (+209%)	80	25	41

and maximal performance. Second, we test target applications for a longer time, until the fuzzers reach a relatively stable state (i.e., the order of fuzzers' performance does not change anymore). Experiments showed that the fuzzers will get stable after testing these applications for 60 hours. So, we tested each application for 60 hours in our experiment.

Experiment environment. We run each fuzzer instance on each target application in the same configuration. More specifically, each instance is run in a virtual machine running Ubuntu 17.04 with one Intel CPU @2.9GHz and 8GB RAM.

4.2 Vulnerability Discovery

Table 1 shows the number of unique vulnerabilities (accumulated in 5 runs) found by 6 different fuzzers in the 19 real world applications. Each application is of the latest version at the time of testing.

In total, AFL, CollAFL, Honggfuzz, VUzzer and Angora has found 21, 34, 18, 6 and 12 vulnerabilities in all applications respectively. GREYONE found 105 unique vulnerabilities in total and covered all vulnerabilities found by other fuzzers. In other words, GREYONE found 209% more vulnerabilities than the second best fuzzer (i.e., CollAFL). Especially, out of these 19 applications, three applications including `nm`, `tiff2pdf` and `libsass` are reported as vulnerable only by GREYONE. In summary, GREYONE significantly outperforms other 5 fuzzers in terms of vulnerability discovery.

The last three columns of Table 1 show the number of vulnerabilities that are previously unknown, known by vendors only and confirmed by CVE respectively. We reported the 105 vulnerabilities we found to upstream vendors, and learned that 25 of them are known by the vendors (but not the public). Among the remaining 80 unknown vulnerabilities, 41 vulnerabilities are confirmed by CVE.

4.3 Unique Crashes Evaluation

In general, the more unique crashes a fuzzer finds, the more vulnerabilities it could find too. Thus, the number of unique crashes is also an important metric for fuzzers. Due to the randomness, we evaluated not only the average but also the maximum number of unique crashes found in 5 runs.

Table 2 shows the detailed evaluation results. GREYONE outperforms all other fuzzers in all applications. Especially in `tiff2pdf`, `nm`, and `libsass`, only GREYONE reported unique crashes and other fuzzers all failed.

Among the 5 runs, GREYONE on average found 716 unique crashes in all applications, which is 501% more than the second best fuzzer (i.e., CollAFL). In the maximum run, GREYONE found 1447 unique crashes in all applications, which is 631% more than the second best fuzzer.

To better examine the efficiency of each fuzzer, we also evaluated the growth trend of unique crashes found by them, as shown in Figure 16 in the Appendix. It shows that, GREYONE had a steady and stronger growth trend on all applications. Furthermore, GREYONE is also the first fuzzer that reported crashes in almost all applications.

4.4 Code Coverage Evaluation

Since a fuzzer can only find vulnerabilities in code that it has explored, code coverage is therefore an important metric for coverage-guided fuzzers.

Table 3 shows the average number of unique paths and edges found by each fuzzer for ten applications. In addition, the improvement of GREYONE compared to the second best fuzzer is also evaluated and showed in the table.

In terms of path coverage, GREYONE outperforms the second best fuzzer by at least 25% in 9 out of ten applications.

Table 2: Number of unique crashes (average and maximum count in 5 runs) found in real world programs by various fuzzers.

Applications	AFL		CollAFL-br		Angora		GREYONE	
	Average	Max	Average	Max	Average	Max	Average	Max
tiff2pdf	0	0	0	0	0	0	6	12
libwpd	0	0	1	3	0	0	21	58
fig2dev	8	12	11	20	0	0	40	79
readelf	0	0	0	0	21	27	28	38
nm	0	0	0	0	0	0	16	72
c++filt	18	30	7	32	0	0	268	575
ncurses	7	18	12	23	0	0	28	37
libsndfile	4	13	8	20	0	0	23	33
libbson	0	0	0	0	0	0	6	12
tiffset	22	46	43	49	0	0	83	122
libsass	0	0	0	0	0	0	8	12
cflow	9	47	17	35	0	0	32	185
nasm	5	15	20	42	6	12	157	212
Total	73	181	119	229	27	39	716 (+501%)	1447 (+631%)

Table 3: Number of unique paths and edges (average in 5 runs) found in real world programs by various fuzzers. Numbers in red are path/edge coverages of the second best fuzzer.

Applications	Path Coverage				Edge Coverage			
	AFL	CollAFL-br	Angora	GREYONE (INC)	AFL	CollAFL-br	Angora	GREYONE (INC)
tiff2pdf	2638	3278	3344	5681(+69.9%)	6261	6776	6820	8250(+20.9%)
readelf	4519	4782	5212	6834(+32%)	6729	6955	7395	8618(+14.5%)
fig2dev	697	764	105	1622(+112%)	934	1754	489	2460(+40.2%)
ncurses	1985	2241	1024	2926(+30.6%)	2082	2151	1736	2787(+28.2%)
libwpd	4113	3856	1145	5644(+37.2%)	5906	5839	4034	7978(+35.1%)
c++filt	9791	9746	1157	10523(+8%)	6387	6578	3684	7101(+8%)
nasm	7506	7354	3364	9443(+25.8%)	6553	6616	4766	8108(+22.5%)
tiffset	1373	1390	1126	1757(+26%)	3856	3900	3760	4361(+11.8%)
nm	2605	2725	2493	4342(+59%)	5387	5526	5235	8482(+53.5%)
libsndfile	911	848	942	1185(+25.8%)	2486	2392	2525	2975(+17.8%)

In the last application `c++filt`, GREYONE outperforms the second best by 8%. In terms of new edge coverage, GREYONE outperforms the second best fuzzer in all applications, on average by 25.5%.

We also evaluated the growth trend of code explored by fuzzers, and presented the path coverage in Fig 13 and edge coverage in Fig 15. It shows that GREYONE has an impressive stronger growth trend than all other fuzzers in all applications.

4.5 Evaluation on LAVA-M

To directly compare the results with other papers, we tested applications in the LAVA-M data set for 24 hours (rather than 60 hours) and repeated 5 times.

Bug finding. Table 4 shows the number of bugs (average in 5 runs) detected by each fuzzer within 24 hours. GREYONE finds 2601 bugs in all applications, including *all listed bugs* in LAVA-M. Moreover, it found 327, 4, 4 and 1 unlisted bugs in these four applications respectively, showing that GREYONE is very effective and much better than other fuzzers.

First, AFL and CollAFL have the worst performance, because they are not sensitive to data flow features and thus unfit for detecting bugs in LAVA-M. Second, Honggfuzz analyzes all operands used in branches, but lacks the ability to isolate untouched branches and lacks efficient seed updating and selection policies. Therefore its evolution speed is slow and the overall efficiency is poor. Third, VUzzer is very slow and can only handle simple constraints (e.g., magic number). Thus it shows minor improvements comparing to AFL. Further, CollAFL-br-LAF integrates the Intel-laf solution, which splits long string comparisons, fit for detecting certain bugs in LAVA-M. Lastly, Angora shows an extraordinary result as well, due to its gradient descent algorithm. However, it may get stuck in local minimum and fail to find certain bugs.

Unique crashes. Figure 5 shows the growth trend of unique crashes found by various fuzzers. Thanks to the accurate taint-guided mutation and stable conformance-based evolution, GREYONE shows a strong and stable growth trend in finding unique crashes. It finds about 1X more unique crashes than the second best fuzzer Angora.

AFL and CollAFL barely could satisfy the complicated path constraints, becasue they are insensitive to data flow features. Interestingly, Anogra shows a fast growth in the beginning and reaches a bottleneck after a few hours. Again, it shows gradient descent is effective at generating interesting test cases. However, it will be trapped soon, due to the inaccuracy of taint and local optimum issue of gradient descent.

4.6 Heuristic Constraints Solving

Note that, GREYONE could bypass a wide range of complicated constraints, by utilizing FTI. In order to further evaluate its effectiveness, we compare it with a state-of-the-art symbolic execution assisted fuzzer QSYM.

To perform fair comparison, we setup similar environments for QSYM and GREYONE. First, we followed the same configuration in the original paper [43] to evaluate QSYM. More specifically, QSYM works together with a master AFL and a slave AFL instance, occupying three CPU cores and 256GB memory. On the other hand, we setup GREYONE to work with

Table 4: The number of bugs found by various fuzzer tools on LAVA-M in 24 hours.

LAVA-M	AFL	CollAFL-br	Honggfuzz	VUzzer	CollAFL-br+laf	QSYM	Angora	GREYONE	Listed bugs
who	0	2	4	49	245	1252(+43)	1438(+95)	2136(+327)	2136
md5sum	1	3	3	12	37	57(+0)	57(+0)	57(+4)	57
base64	2	2	6	15	44	44(+4)	44(+4)	44(+4)	44
uniq	1	1	4	24	21	28(+1)	28(+1)	28(+1)	28

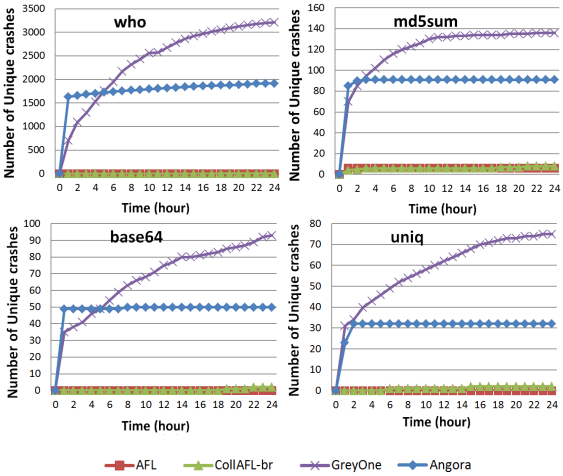

Figure 5: The growth trend of unique crashes found in LAVA-M by AFL, CollAFL, Angora and GREYONE.

a slave AFL by simply sharing their seed queues, occupying only two CPU cores and 8GB memory.

Table 5 shows the head-to-head comparison results, in 5 runs with 60 hours each time. Although GREYONE takes fewer computing resources, it outperforms QSYM in terms of both code coverage and vulnerabilities discovery. On average, GREYONE found 1.2X unique paths, 1.12X edges, 2.15X unique crashes and 1.52X vulnerabilities than QSYM.

To further demonstrate the effectiveness of constraints solving, we tracked the growth trend of paths coverage and presented in Figure 6. We could find GREYONE cover more paths in a faster pace than QSYM in most subjects.

According to the above evaluation, the heuristic constraint solving capability provided by GREYONE outperforms symbolic constraint solver when applied to hybrid fuzzing.

5 Further Analysis

We further evaluated GREYONE's ability of data flow analysis and the outcome of applying such data flow features to fuzzing, to better understand the improvements of GREYONE.

5.1 Performance of FTI

Our taint analysis engine FTI provides support for further taint-guided mutation and conformance-guided evolution, playing an important role in GREYONE. In this section, we evaluated the efficiency and performance of FTI.

5.1.1 Completeness of Taint Inference

As aforementioned, FTI is sound and has no over-taint issues. However, it may have under-taint issues due to its incomplete testing in the pilot fuzzing. We hereby evaluated the under-taint issues FTI is facing.

Note that, it is infeasible to get the ground truth of the accurate taint information, even if the source code is given, due to challenges like implicit data flows and external dependencies. As a result, we directly compare FTI with another dynamic taint analysis (DTA) engine, to roughly estimate under-taint.

Experiment Setup. There are several taint analysis engines available [2, 20], we chose DFSan [2] as the DTA engine to compare with, since it is the official engine shipped with the LLVM [22] compilation framework and has good runtime performance and platform support.

As aforementioned, solutions like DFSan not only suffer from implicit data flows, but also external dependencies. For example, if an external library is not processed with DFSan, the taint propagation will be broken once it flows into the library. To mitigate this issue, we built taint models for all external libraries used in the experiment. Therefore, DFSan could get more taint information than its default configuration.

Then, we built a variation of GREYONE, named as GREYONE-DTA by replacing its taint analysis engine with DFSan. Further we tested GREYONE and GREYONE-DTA on 11 real world applications and 4 applications from LAVA-M. For each application, we randomly selected hundreds of unique program paths that have been explored by both GREYONE and GREYONE-DTA. Then we examined all untouched branches in these paths, and counted the number of untouched branches that are related to input bytes (i.e., tainted).

Figure 7 shows the proportion of tainted untouched branches reported by GREYONE of version FTI and DTA. Note that, FTI has no over-taint issues, but DTA may have over-taint issues (e.g., due to wrong taint propagation in `XOR` instructions etc.). From the figure, we can learn that:

- DTA still has serious under-taint issues in all applications, even though we have mitigated some (caused by external dependencies). All the tainted untouched branches reported by FTI-only are missed by DTA. Most of these under-taint issues are caused by implicit data flows.
- FTI has fewer under-taint issues. It also finds much more taint (without over-taint) than DTA, even if DTA could have over-claimed. For example, DTA could only identify 25% of taint reported by FTI in the application `fig2dev`. On average, FTI could find 1.3X times more tainted untouched branches than DTA.

Table 5: Number of unique paths, unique edge, unique crashes (average count in 5 runs with 60 hours each time) and total vulnerabilities (5 runs with 60 hours each time) found in real world programs by QSYM-* (QSYM+master AFL+ slave AFL) and GREYONE-* (GREYONE +slave AFL).

Applications	Average Unique Paths		Average Unique Edges		Average Unique Crashes		Total Vulnerabilities	
	QSYM-*	GREYONE-*	QSYM-*	GREYONE-*	QSYM-*	GREYONE-*	QSYM-*	GREYONE-*
Readelf	9028	12312(+36.38%)	7822	8847(+13.10%)	46	77	4	4
Nm	4218	5822(+38.03%)	6773	8599(+26.96%)	3	18	1	2
C++filt	10988	12122(+10.32%)	6898	7155(+3.73%)	158	299	4	4
Tiff2pdf	4856	5698(+17.34%)	7431	8088(+8.84%)	0	3	0	2
Tiffset	1897	2205(+16.24%)	4285	4404(+2.78%)	25	66	2	2
Libwpd	8279	10589(+31.27%)	9947	11702(+17.64%)	12	24	1	2
libsndfile	1375	1650(+20%)	2691	3033(+12.71%)	32	46	1	2
Fig2dev	1218	1616(+32.68%)	1843	2241(+21.60%)	15	38	6	10
Nasm	9184	9529(+3.76%)	7433	8104(+9.03%)	87	231	8	11
libncurses	2837	3291(+16%)	2749	2950(+7.31%)	36	88	3	5
Average Improvement	-	+20.34%	-	+12.53%	-	+115%	-	+52%

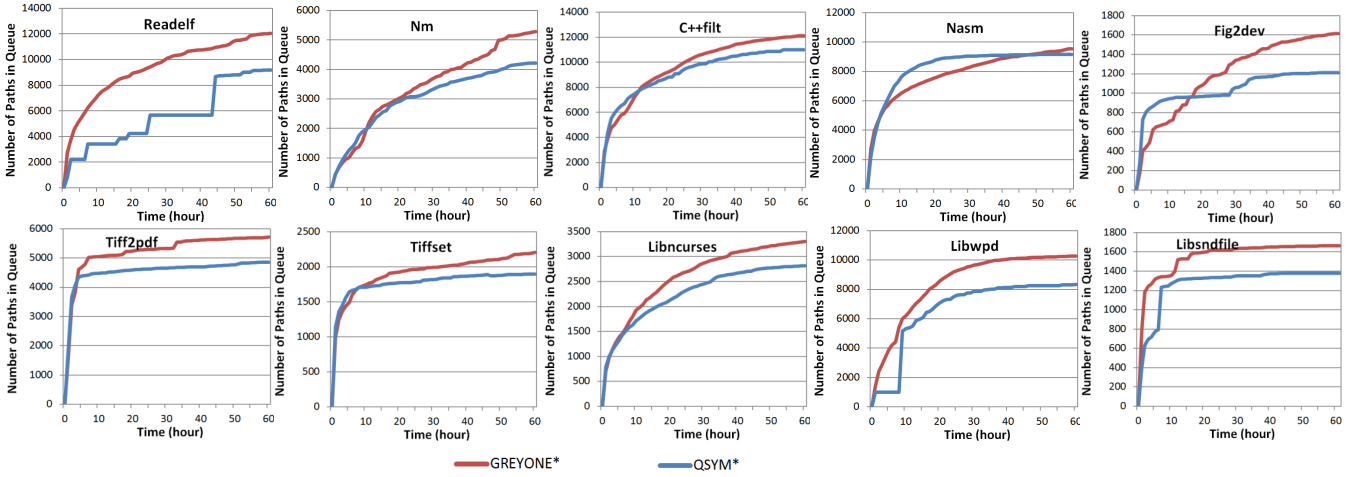

Figure 6: The growth trend of number of unique paths (average of 5 runs) detected by QSYM-* and GREYONE-*.

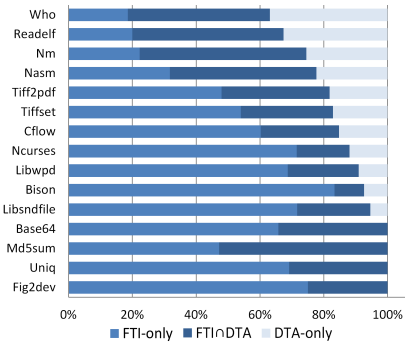

Figure 7: Proportion of tainted untouched branches reported by FTI-only, DTA-only and both FTI and DTA.

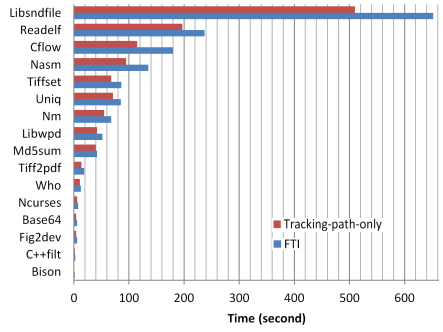

Figure 8: Average speed of analyzing one seed by FTI.

5.1.2 Overhead of Taint Inference

As aforementioned, for each seed, FTI first performs byte-level mutation to generate new test cases. It then tests the target applications and tracks the code coverage. During testing, FTI monitors the value changes and infers taint for all untouched branches in the path explored by the original seed.

Figure 8 shows the average speed of analyzing one seed by FTI. The bar named `tracking-path-only` represents the time used for byte-level mutation and fuzzing. The bar `FTI` also includes the time of taint inference including value monitoring. It shows that taint inference introduces less than 25% overheads. Figure 9 further shows the time of inferring taint for one branch instruction in the path. On average, FTI spends 0.15 seconds on inferring taint for one branch instruction..

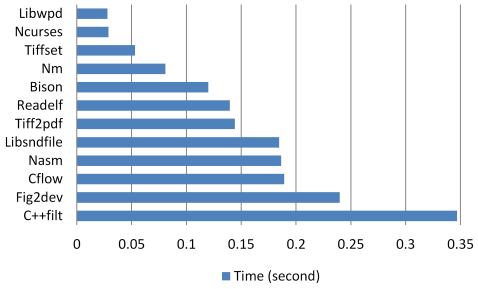

Figure 9: Average speed of inferring taint for one branch instruction, given input seeds of 1KB size.

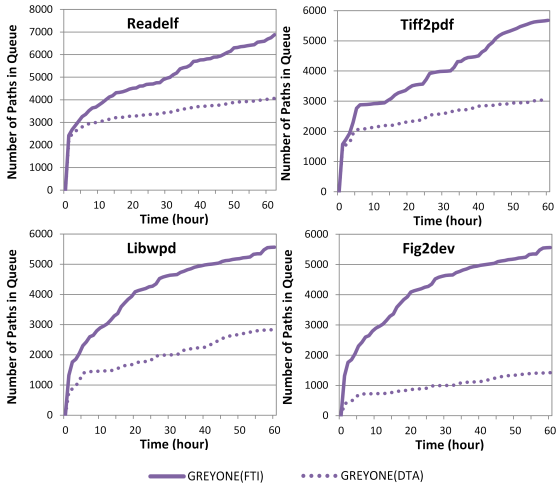

Figure 10: Code coverage improvement brought by FTI.

5.2 Improvements Breakdown

GREYONE adopts two major data flow features, i.e., taint and constraint conformance, and several schemes to improve the efficiency of fuzzing. We hereby breakdown the improvements of each scheme.

a) Taint Inference. Figure 10 shows the code coverage brought by GREYONE and GREYONE-DTA, which replaces the taint inference engine FTI with another engine DFSan. It shows that, on average, FTI could double the code coverage on all targets, comparing to GREYONE-DTA. Thus, our taint analysis engine FTI is useful.

b) Bytes prioritization. GREYONE uses taint to guide mutation, by prioritizing input bytes to mutate, and determine the way to mutate. We hereby measured the improvements brought by byte prioritization. As shows in Table 6, after disabling bytes prioritization, GREYONE-BP could explore much less code and find fewer vulnerabilities on all applications. On average, it has 14% fewer unique paths and 42% fewer unique crashes than GREYONE.

We further tracked the growth trend of unique paths and unique crashes. Figure 11 shows that, in terms of code coverage, with byte prioritization, GREYONE could find about 20% more paths in applications `tiff2pdf` and `libwpd`. In terms of unique crashes, with byte prioritization, GREYONE could find unique crashes faster, and find much more. Especially, when testing the application `tiff2pdf`, GREYONE could not find any crashes in 60 hours if byte prioritization is turned off.

c) Conformance-guided Evolution. GREYONE utilizes conformance to guide the evolution direction of fuzzing. We also evaluated the improvements of this scheme, in a way similar to byte prioritization. As shows in Table 6, after disabling conformance-guided, GREYONE-CE explores much less code and find fewer vulnerabilities on all applications, even worse than GREYONE-BP. On average, it has 21.9% fewer unique paths and 63.2% fewer unique crashes than GREYONE.

Specially, without conformance-guided evolution, GREYONE found 30% fewer paths in all applications, and failed to find any unique crashes in Tiff2pdf and libwpd.

d) Selective execution. The advantage of selective mechanism is to avoid select the correspondent instance to execute when the new seed is mutated too many bytes or has low probability to generate better conformance. By taking this strategy, the most intuitive effect to fuzzing is to improve the overall execution speed. For showing the promotion, we conducted two selective mode in GREYONE, one was the default set, the other was only to select the instance with monitoring conformance to execute. As shown in Fig 12, we tested 14 subjects and evaluated the average execution speed on each subject. Comparing to AFL, GREYONE with selective mechanism can reach a speed at over 80%, while GREYONE without selective mechanism could only reach a speed at less than 65%.

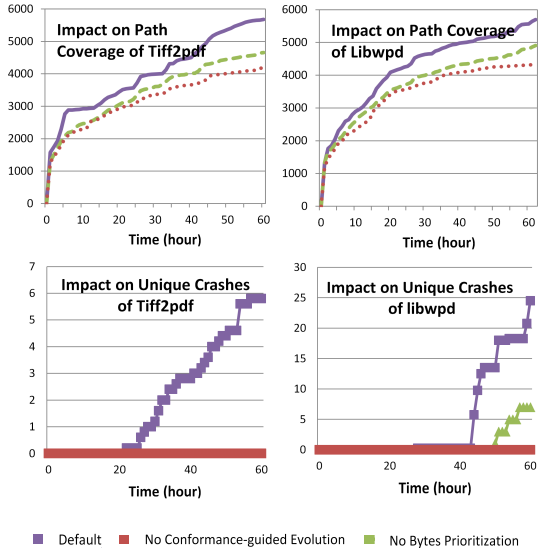

Figure 11: Improvements brought by *byte prioritization* and *conformance-guided evolution*, in terms of code coverage and unique crashes found in two applications.

Table 6: Number of unique paths and crashes (average in 5 runs with 60 hours one run) found in real world programs by GREYONE, GREYONE-CE and GREYONE-BP, where GREYONE-CE is the mode of GREYONE disabling conformance-guided evolution and GREYONE-BP is the mode of GREYONE disabling bytes prioritization.

Applications	Unique Paths			Unique Crashes		
	GREYONE	GREYONE-CE	GREYONE-BP	GREYONE	GREYONE-CE	GREYONE-BP
Readelf	6834	6222(-9%)	5757(-15.8%)	28	21(-25%)	25(-10.7%)
Nm	4342	3432(-21%)	3886(-10.5%)	16	4(-75%)	7(-56.3%)
C++filt	10523	9870(-6.2%)	9932(-5.6%)	268	127(-52.6%)	225(-16%)
Tiff2pdf	5681	4107(-27.8%)	4598(-19%)	6	0(-100%)	0(-100%)
Tiffset	1757	1345(-23.4%)	1434(-18.4%)	83	28(-66.3%)	49(-41%)
Libwpd	5644	4220(-25.2%)	4982(-11.7%)	21	0(-100%)	7(-66%)
libsndfile	1185	1069(-10%)	1081(-8.2%)	23	7(-69.6%)	9(-60.9%)
Fig2dev	1622	999(-38.4%)	1211(-25.3%)	40	24(-40%)	33(-17.5%)
Nasm	9443	6578(-30.3%)	7979(-15.5%)	157	28(-82.2%)	79(-49.7%)
libncurses	2926	2112(-27.8%)	2543(-13%)	28	22(-21.4%)	25(-10.7%)
Average Reduction	-	-21.9%	-14.3%	-	-63.2%	-42.9%

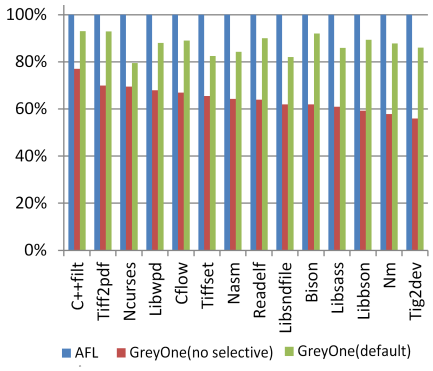

Figure 12: The speed impact brought by selective execution in GREYONE (60 hours).

6 Related Work

Evolutionary mutation-based fuzzing achieved a great success in practice, due to its scalability and efficiency. The representative solution AFL [44] takes achieving higher code coverage as evolution direction, and mutates seeds in a nearly random manner. Many other solutions, including taint analysis, have been proposed to improve mutation-based fuzzing.

6.1 Taint Inference

Taint analysis is a fundamental technique for many applications including fuzzing. Traditional taint analysis solutions [2, 20] heavily rely on manual efforts of compose taint propagation rules for each instruction, and suffer from serious under-taint and over-taint problems.

Improvements to traditional taint analysis. Many alleviated schemes are proposed to mitigate the inaccuracy issue for traditional taint analysis. Dytan [11] keeps track of indirect taint propagation to mitigate the under-taint issue, but brings lots of false positives. DTA++ [19] locates implicit control flow branch and diagnose under-taint using offline symbolic execution. However, it suffers from solving complicated conditions and high performance overheads. TAINTINDUCE [46] adopts a testing-based solution to infer taint propagation rules automatically. But it is very heavy-weight, and cannot solve the inaccuracy issues.

Mutation-based inference. Some recent works proposed mutation-based taint inference which have better performance in certain applications. Sekar [31] adopts black-box testing and leverages predefined mutation rules to infer taint, able to detect injection attacks. MutaFlow [26] monitors changes of security-sensitive APIs by mutating sensitive source APIs, able to detect vulnerable information flow. These two focus on local program behaviors and are limited to information flow detection. In fuzzing applications, REDQUEEN [4] uses random mutation to colorize inputs, to infer taint related to direct copy of inputs. Fairfuzz [24] and ProFuzzer [42] monitor the pattern of control flow changes among multiple runs, to infer partial type of mutated bytes. None of these solutions have ever considered the variables' value changes after mutation, Thus, they all fail to provide accurate taint information.

In this paper, we propose a fuzzing-driven taint inference solution FTI. We perform a systematic byte-level mutation to perform a pilot fuzzing. During fuzzing, we monitor variables' value changes and infer taint attributes accordingly. This solution is automated, lightweight and more accurate.

6.2 Seed Mutation

Many studies [10, 13, 30, 44] have shown that, seed mutation is one of the most hot and hard direction to increase the efficiency and accuracy of fuzzing. Many approaches are proposed to try to solve how and where to mutate.

a) Static-analysis-based optimization. Steelix [25] and Laf-intel-pass [1] statically decompose those long constant comparisons into multiple shorter comparisons. So that the dumb random fuzzer could satisfy the path constraints with a much higher probability. However, it brings too many semantic-equivalent paths to explore, and cannot handle non-constant comparisons. SYMFUZZ [8] leverages static symbolic analysis to detect dependencies among input bits, and uses it to compute an optimal mutation ratio. However, this process is slow, and the calculated dependency between bits do not show many improvements for mutation.

b) Learning-based model. Rajpal et.al. [29] presents a RNN-based model to predict best locations to mutate in seeds,

based on the history mutations and their corresponding code coverage feedback. Konstantin et.al. [7] uses deep reinforcement learning to model the fuzzing loop and choose the best mutation actions in the following fuzzing iteration. These solutions are in early stage and have not shown significant improvements yet. NEUZZ [35] identifies the significance of program smoothing and uses an incremental learning technique to guide mutation.

c) Symbolic-based solution. This type of solutions essentially utilize symbolic execution to solve the complicated path constraints that are hard to be satisfied by mutation-based fuzzing. Driller [37] periodically picks paths that are stuck in mutation-based fuzzing, and uses symbolic execution to solve the constraints of those paths. QSYM [43] ports symbolic execution to native X86 instructions and relaxes the path constraints to solve, providing a better analysis performance and reducing the speed of constraint solving. DigFuzz [45] designs a probabilistic path prioritization model to quantify each path's difficulty and prioritize them for concolic execution. All of these symbolic-based solution cannot scale to large applications due to the open challenge of constraint solving.

d) Taint-based mutation. Several fuzzers utilize taint to guide mutations. Dowser [17] and BORG [27] use taint to locate buffer boundary violations and buffer over-read vulnerabilities respectively. BuzzFuzz [15] uses DTA to track the regions of external seed inputs that affect sensitive library or system calls. TaintScope [40] leverages fine-grained DTA to identify checksum branch. VUzzer [30] is able to track branches that compare variables against constants, e.g., magic numbers, and guides the mutation accordingly. Angora [10] performs shape inference and gradient descent computation based on DTA. These solutions suffer from inaccurate taint, limiting the efficiency in complicated programs.

In addition, the high overhead of DTA greatly limits the application of DTA in large applications. Among the light-weight taint-guide mutation solutions, Fairfuzz [24] and Profuzzer [42] could not obtain accurate taint attributes of variables, inefficient at exploring hard-to-reach branches. In addition, they would repeatedly mutate some input bytes, even if the relevant branches have already been explored, since they are insensitive to branch states. REDQUEEN [4] focus on identifying direct copy of inputs and branches use them, unable to handle the prevalent uses of indirect copy of inputs.

Our solution GREYONE utilizes the lightweight and sound taint inference solution FTI to get more taint attributes (without over-taint) as well as the precise relationship between input offsets and branches, to prioritize which branch to explore and which bytes to mutate, as well as determine how to precisely mutate them.

6.3 Seed Updating and Selection

Seed updating and selection could adjust the evolution direction of fuzzing. A good solution would improve the efficiency of fuzzers in finding more code and bugs [28] and in moving towards potentially vulnerable target code [5, 9, 39].

Few works focus on seed updating, but many seed selection solutions are proposed in the past years. These solutions in general collect more and more auxiliary control flow information to guide the seed selection. At the beginning, AFL [44] prioritizes those seeds with smaller size and shorter execution time, to generate more test cases in a given time period. Then, AFLFAST [6] points out the importance of seed selection, and prioritizes seeds that are rarely picked to mutate and that explore cold paths. From then on, kinds of control flow characteristics are used to guide seed selection, e.g., by prioritizing deeper path [30] or untouched neighbour branches [14].

However, these solutions did not consider any data flow features, and thus are inefficient at exploring paths with complicated constraints. Honggfuzz [38] and LibFuzzer [32] took a weak data flow feature to guide seed selection. More specifically, they evaluate the distance between operands of all branches, and use it to guide seed selection.

GREYONE improves this strategy by evaluating the constraint conformance on all tainted untouched branches only. It also utilizes a novel two-dimension seed queue structure to provide support for efficient seed updating and selection. It is able to avoid the local minimum problem facing by the learning-based solution used in Angora [10]. Further, GREYONE applies a novel on-the-fly mutation rebase to further accelerate the evolution of fuzzing.

6.4 Performance Optimization

Performance is an important factor of efficient fuzzing. Several solutions have been proposed to improve the fuzzing performance, by boosting the parallel execution [41] or instrumentation [18, 36]. The recent work Untracer [36] removes unnecessary instrumentation in basic blocks that have been explored and reduces the overhead. GREYONE also optimizes the instrumentation, to select more light-weight testing mode on demand, and switch between different fuzzing mode, to improve the speed of fuzzing.

7 Conclusion

In this paper, we propose a novel data flow sensitive fuzzing solution GREYONE. It infers taint during the fuzzing process by monitoring variable value changes, and further guides seed mutation with the inferred taint. It also applies a data flow feature *conformance* to tune the evolution direction of fuzzing, driving the fuzzer to quickly reach unexplored branches and trigger potential vulnerabilities. It outperforms various state-of-the-art fuzzers in terms of both code coverage and vulnerability discovery, while its taint analysis is more lightweight and accurate than others.

References

[1] Circumventing Fuzzing Roadblocks with Compiler Transformations. https://lafintel.wordpress.com/2016/08/15/circumventing-fuzzing-roadblocks-with-compiler-transformations/.

[2] Dataflowsanitizer. https://clang.llvm.org/docs/DataFlowSanitizerDesign.html.

[3] Utilities for automated crash sample processing/analysis. https://github.com/rc0r/afl-utils.

[4] ASCHERMANN, C., SCHUMILO, S., BLAZYTKO, T., GAWLIK, R., AND HOLZ, T. Redqueen: Fuzzing with input-to-state correspondence. In *NDSS* (2019). To appear.

[5] BOHME, M., PHAM, V.-T., NGUYEN, M.-D., AND ROYCHOUDHURY, A. Directed greybox fuzzing. In *CCS* (2017).

[6] BÖHME, M., PHAM, V.-T., AND ROYCHOUDHURY, A. Coverage-based greybox fuzzing as markov chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (2016), ACM, pp. 1032–1043.

[7] BÖTTINGER, K., GODEFROID, P., AND SINGH, R. Deep reinforcement fuzzing. *arXiv preprint arXiv:1801.04589* (2018).

[8] CHA, S. K., WOO, M., AND BRUMLEY, D. Program-adaptive mutational fuzzing. In *Security and Privacy (SP), 2015 IEEE Symposium on* (2015), IEEE, pp. 725–741.

[9] CHEN, H., XUE, Y., LI, Y., CHEN, B., XIE, X., WU, X., AND LIU, Y. Hawkeye: towards a desired directed grey-box fuzzer. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (2018), ACM, pp. 2095–2108.

[10] CHEN, P., AND CHEN, H. Angora: Efficient fuzzing by principled search. *arXiv preprint arXiv:1803.01307* (2018).

[11] CLAUSE, J., LI, W., AND ORSO, A. Dytan: a generic dynamic taint analysis framework. In *Proceedings of the 2007 international symposium on Software testing and analysis* (2007), ACM, pp. 196–206.

[12] DOLAN-GAVITT, B., HULIN, P., KIRDA, E., LEEK, T., MAMBRETTI, A., ROBERTSON, W., ULRICH, F., AND WHELAN, R. Lava: Large-scale automated vulnerability addition. In *Security and Privacy (SP), 2016 IEEE Symposium on* (2016), IEEE, pp. 110–121.

[13] EDDINGTON, M. Peach fuzzing platform. *Peach Fuzzer* (2011), 34.

[14] GAN, S., ZHANG, C., QIN, X., TU, X., LI, K., PEI, Z., AND CHEN, Z. Collafl: Path sensitive fuzzing. In *2018 IEEE Symposium on Security and Privacy (SP)* (2018), IEEE, pp. 679–696.

[15] GANESH, V., LEEK, T., AND RINARD, M. Taint-based directed whitebox fuzzing. In *Proceedings of the 31st International Conference on Software Engineering* (2009), IEEE Computer Society, pp. 474–484.

[16] GOGUEN, J. A., AND MESEGUER, J. Security policies and security models. In *1982 IEEE Symposium on Security and Privacy* (1982), IEEE, pp. 11–11.

[17] HALLER, I., SLOWINSKA, A., NEUGSCHWANDTNER, M., AND BOS, H. Dowsing for overflows: a guided fuzzer to find buffer boundary violations. In *USENIX Security Symposium* (2013), pp. 49–64.

[18] HSU, C.-C., WU, C.-Y., HSIAO, H.-C., AND HUANG, S.-K. Instrim: Lightweight instrumentation for coverage-guided fuzzing. In *Symposium on Network and Distributed System Security (NDSS), Workshop on Binary Analysis Research* (2018).

[19] KANG, M. G., MCCAMANT, S., POOSANKAM, P., AND SONG, D. Dta++: dynamic taint analysis with targeted control-flow propagation. In *NDSS* (2011).

[20] KEMERLIS, V. P., PORTOKALIDIS, G., JEE, K., AND KEROMYTIS, A. D. libdft: Practical dynamic data flow tracking for commodity systems. In *Acm Sigplan Notices* (2012), vol. 47, ACM, pp. 121–132.

[21] KLEES, G., RUEF, A., COOPER, B., WEI, S., AND HICKS, M. Evaluating fuzz testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (2018), ACM, pp. 2123–2138.

[22] LATTNER, C. LLVM related publications. Official LLVM web site. Retrieved on 2010-12-04. http://llvm.org.

[23] LEE, B., SONG, C., KIM, T., AND LEE, W. Type casting verification: Stopping an emerging attack vector. In *USENIX Security Symposium* (2015), pp. 81–96.

[24] LEMIEUX, C., AND SEN, K. Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering* (2018), ACM, pp. 475–485.

[25] LI, Y., CHEN, B., CHANDRAMOHAN, M., LIN, S.-W., LIU, Y., AND TIU, A. Steelix: program-state based binary fuzzing. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (2017), ACM, pp. 627–637.

[26] MATHIS, B., AVDIIENKO, V., SOREMEKUN, E. O., BÖHME, M., AND ZELLER, A. Detecting information flow by mutating input data. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering* (2017), IEEE Press, pp. 263–273.

[27] NEUGSCHWANDTNER, M., MILANI COMPARETTI, P., HALLER, I., AND BOS, H. The borg: Nanoprobing binaries for buffer overreads. In *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy* (2015), ACM, pp. 87–97.

[28] PETSIOS, T., ZHAO, J., KEROMYTIS, A. D., AND JANA, S. Slowfuzz: Automated domain-independent detection of algorithmic complexity vulnerabilities. In *Conf. on Computer and Communication Security* (2017).

[29] RAJPAL, M., BLUM, W., AND SINGH, R. Not all bytes are equal: Neural byte sieve for fuzzing. *CoRR abs/1711.04596* (2017).

[30] RAWAT, S., JAIN, V., KUMAR, A., AND BOS, H. VUzzer: Application-aware Evolutionary Fuzzing. In *Network and Distributed System Security Symposium* (2017).

[31] SEKAR, R. An efficient black-box technique for defeating web application attacks. In *NDSS* (2009).

[32] SEREBRYANY, K. Continuous fuzzing with libfuzzer and addresssanitizer. In *Cybersecurity Development (SecDev), IEEE* (2016), IEEE, pp. 157–157.

[33] SEREBRYANY, K. Oss-fuzz - google's continuous fuzzing service for open source software.

[34] SEREBRYANY, K., BRUENING, D., POTAPENKO, A., AND VYUKOV, D. Addresssanitizer: A fast address sanity checker. In *the 2012 USENIX Annual Technical Conference* (2012), pp. 309–318.

[35] SHE, D., PEI, K., EPSTEIN, D., YANG, J., RAY, B., AND JANA, S. Neuzz: Efficient fuzzing with neural program smoothing. In *IEEESP* (2019). To appear.

[36] STEFAN NAGY, M. H. Full-speed fuzzing: Reducing fuzzing overhead through coverage-guided tracing. In *IEEESP* (2019). To appear.

[37] STEPHENS, N., GROSEN, J., SALLS, C., DUTCHER, A., WANG, R., CORBETTA, J., SHOSHITAISHVILI, Y., KRUEGEL, C., AND VIGNA, G. Driller: Augmenting fuzzing through selective symbolic execution. In *NDSS* (2016), vol. 16, pp. 1–16.

[38] SWIECKI, R. Honggfuzz. *Available online a t: http://code. google. com/p/honggfuzz* (2016).

[39] WANG, S., CHANG NAM, J., AND TAN, L. Qtep: Qulity-aware test case prioritization. In *ESEC/FSE 2017 Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (2017).

[40] WANG, T., WEI, T., GU, G., AND ZOU, W. Taintscope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection. In *IEEE Symposium on Security and Privacy* (2010).

[41] XU, W., KASHYAP, S., MIN, C., AND KIM, T. Designing new operating primitives to improve fuzzing performance. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2017), CCS '17, ACM, pp. 2313–2328.

[42] YOU, W., WANG, X., MA, S., HUANG, J., ZHANG, X., WANG, X., AND LIANG, B. Profuzzer: On-the-fly input type probing for better zero-day vulnerability discovery. In *IEEE Security and Privacy* (2019), IEEE.

[43] YUN, I., LEE, S., XU, M., JANG, Y., AND KIM, T. {QSYM}: A practical concolic execution engine tailored for hybrid fuzzing. In *27th {USENIX} Security Symposium ({USENIX} Security 18)* (2018), pp. 745–761.

[44] ZALEWSKI, M. American fuzzy lop. http://lcamtuf.coredump.cx/afl/.

[45] ZHAO, L., DUAN, Y., YIN, H., AND XUAN, J. Send hardest problems my way: Probabilistic path prioritization for hybrid fuzzing. In *NDSS* (2019). To appear.

[46] ZHENG LEONG CHUA, Y. W. Inferring taint rules without architectural semantics. In *NDSS* (2019). To appear.

A APPENDIX

Due to the space limit, we present some of the evaluation results here.

A.1 Growth Trend of Code Coverage

In this section, we present the evaluation result of the code coverage growth trend and the effects of randomness.

Code Coverage. Figure 13 and Fig. 15 show the average growth trend of paths and edges detected by each fuzzer in five runs. It shows that GREYONE has a stronger growth trend

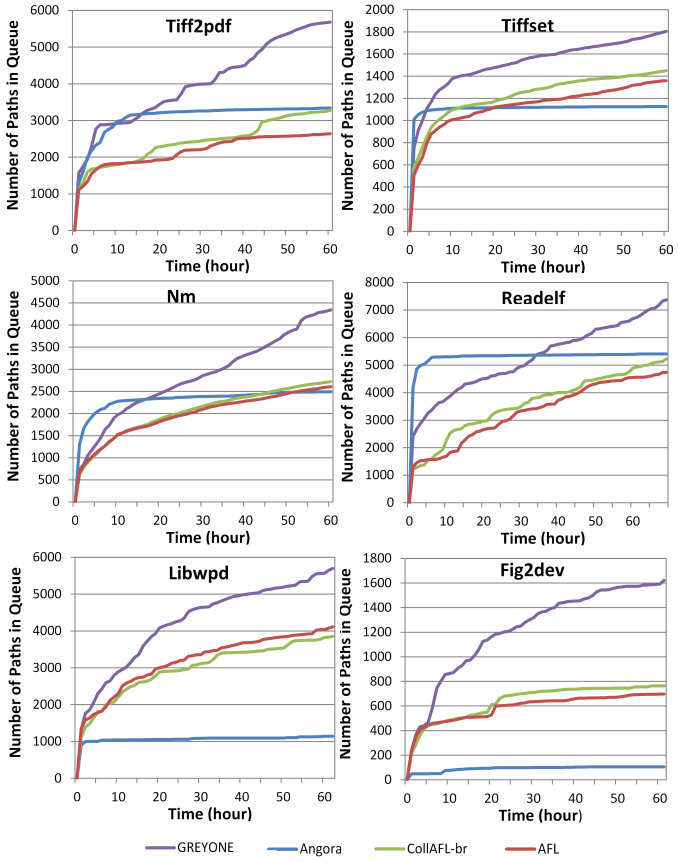

Figure 13: The growth trend of number of unique paths (average in 5 runs) detected by AFL, CollAFL-br, Angora and GREYONE.

than other fuzzers in all applications. Unlike other fuzzers, GREYONE keeps a steady growth trend for a long time.

For example, in the application `readelf`, GREYONE fell behind Angora at the beginning. But it caught up with Angora at 40 hours, and maintained a strong and steady growth trend, far surpassing Angora finally.

Among other fuzzer tools, Angora could achieve high code coverage in a very short time in some applications, e.g., `readelf` and `nm` It proves that its gradient descent based mutation is effective. However, it may fall into local minimum soon, leading to very poor code coverage on most applications, e.g., `libwpd`, `fig2dev`, `libncurses`, abd `c++filt`.

Randomness. As shown in Fig 14, the randomness in fuzzing does not affect the conclusion, the worst run of GREYONE still shows better code coverage than the best run of other fuzzers.

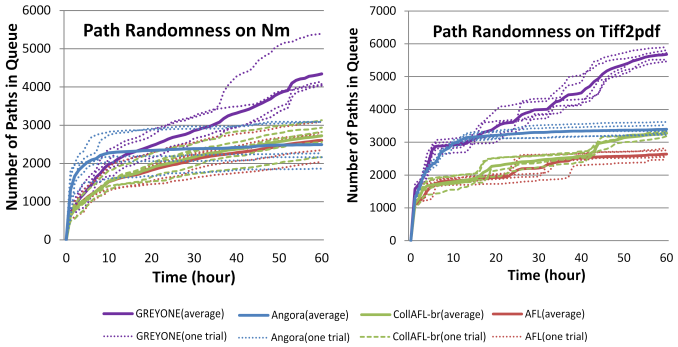

Figure 14: Path randomness.

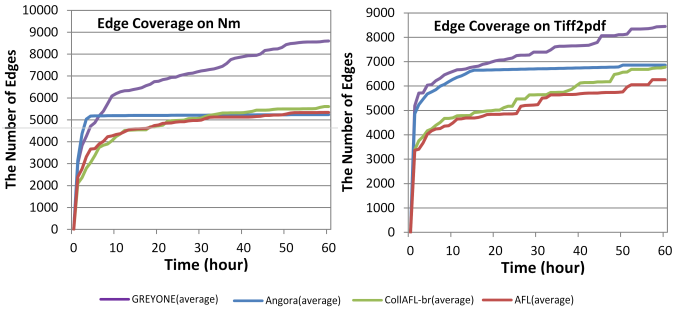

Figure 15: Edge coverage.

A.2 Growth Trend of Unique Crashes

In this section, we present the growth trend of unique crashes and the effects of randomness.

Unique Crashes. As shown in Fig 16, GREYONE has a strong growth trend on each application. Comparing to other fuzzers, GREYONE could find more unique crashes in almost all applications. It also finds crashes faster than other fuzzers in all applications except `readelf`. Similar to growth trend of paths, Angora could find more crashes than GREYONE in earlier stage on the subject `readelf`, but is surpassed by GREYONE after 50 hours.

Randomness. The number of unique crashes is more sensitive to randomness than code coverage, because crashes are rare comparing to program path. However, we can see that worst run of GREYONE in general still shows better code coverage than the best run of other fuzzers.

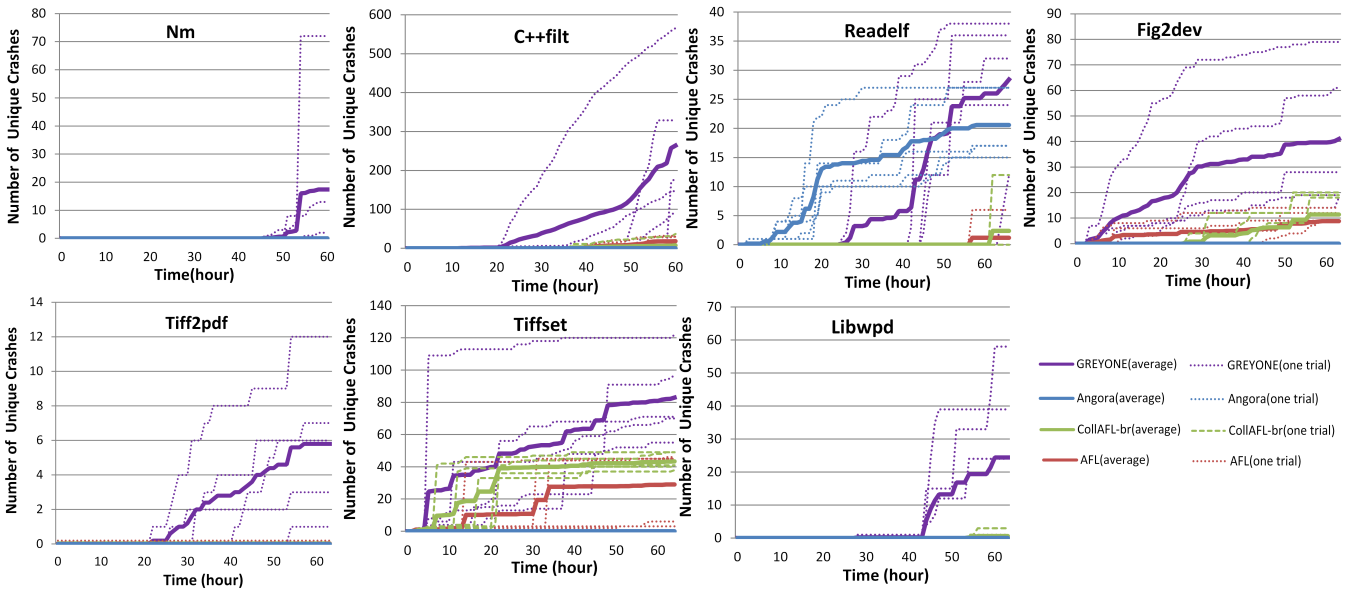

Figure 16: The growth trend of number of unique crashes (average and each of 5 runs) detected by AFL, CollAFL-br, Angora and GREYONE.