Not All Seeds Are Important: Fuzzing Guided by Untouched Edges

Chen Xie 1, Peng Jia 1,*, Pin Yang 1, Chi Hu 2, Hongbo Kuang 1, Genzuo Ye 1 and Xuanquan Hong 1

1 School of Cyber Science and Engineering, Sichuan University, Chengdu 610207, China; 
xiechen1@stu.scu.edu.cn (C.X.); yangpin@scu.edu.cn (P.Y.); hongbokuang@stu.scu.edu.cn (H.K.); 
yegenzuo@stu.scu.edu.cn (G.Y.); leehung@stu.scu.edu.cn (X.H.)
2 China Academy of Engineering Physics (CAEP), Mianyang 621900, China; huchi16@nudt.edu.cn
* Correspondence: pengjia@scu.edu.cn

Abstract: Coverage-guided greybox fuzzing (CGF) has become the mainstream technology used in the field of vulnerability mining, which has been proven to be effective. Seed scheduling, the process of selecting seeds from the seeds pool for subsequent fuzzing iterations, is a critical component of CGF. While many seed scheduling strategies have been proposed in academia, they all focus on the explored regions within programs. In response to the inefficiencies of traditional seed scheduling strategies, which often allocate resources to ineffective seeds, we introduce a novel seed scheduling strategy guided by untouched edges. The strategy generates the optional seed set according to the information on the untouched edges. We also present a new instrumentation method to capture unexplored areas and guide the fuzzing process toward them. We implemented the prototype UntouchFuzz on top of American Fuzzy Lop (AFL) and conducted evaluation experiments against the most advanced seed scheduling strategies. Our results demonstrate that UntouchFuzz has improved in code coverage and unique vulnerabilities. Furthermore, the method proposed is transplanted into the fuzzer MOpt, which further proves the scalability of the method. In particular, 13 vulnerabilities were found in the open-source projects, with 7 of them having assigned CVEs.

Keywords: vulnerability mining; greybox fuzzing; seed scheduling

1. Introduction

Fuzzing is a prevalent and effective automated software testing method that has already found numerous vulnerabilities in real-world applications [1–7]. Fuzzers efficiently explore the input space of the program under test, operating at nearly raw execution speeds, with the aim of identifying specific inputs that can provoke program crashes or anomalous behaviors. However, the input space of most real-world programs is so large that it is difficult to fully explore. Moreover, vulnerabilities are sparse in an application, with only certain specific inputs capable of triggering vulnerabilities [8].

American Fuzzy Lop (AFL) [9], one of the most popular and widely used coverage-guided greybox fuzzers in both academia and industry, is an efficient fuzzing tool for file applications and has already discovered many high-risk vulnerabilities across various projects. AFL employs a mutation-based fuzzing approach by mutating the binary data of the seed file to find test cases that improve coverage or trigger crashes. Research [10] indicates that the performance of mutation-based fuzzers depends on seed scheduling, essentially determining the prioritization of which seed to mutate.

The main challenge in seed scheduling is to determine which seeds in the corpus are more likely to explore new code space in the program when mutated. From the perspective of code coverage, the main role of seed scheduling is to prioritize those seeds that are more promising to trigger new code coverage after being mutated. AFL, for example, utilizes a greedy algorithm to maintain an optimal seed set that covers all explored edges. However, seeds related to validation check edges are also added to the seed set by mistake, and fuzzing these seeds will waste a lot of computational overhead [11].
In the field of seed scheduling, many scholars have conducted in-depth research. Classic seed scheduling strategies mainly guide seed scheduling and prioritization through the distribution of edge coverage or path coverage throughout the fuzzing process. For example, AFLFast [12] gives priority to those seeds likely to trigger low-frequency paths. FairFuzz [13] prioritizes those triggering rare edges, while EcoFuzz [14] prioritizes those based on the seed’s self-transition probability. However, the distribution of coverage information mentioned above is related to the control flow graph (CFG) of the program. Consequently, the performance of such fuzzers can vary across programs with different CFGs.

To decouple the strong correlation between seed scheduling strategies and target programs, SLIME [15] classifies seeds by constructing multiple property queues and employs the upper confidence bound variance (UCB-V) algorithm to select the optimal seed queue and then fuzzing the seeds in that queue. Furthermore, Alphuzz [16] considers seed interdependencies and uses a Monte Carlo search tree approach for seed scheduling.

However, the existing methods mentioned above neglect to focus on the unexplored regions within the control flow graph (CFG) of the program. For instance, consider a seed s whose execution path does not contain any untouched edges, but a coverage-guided fuzzer still marks s as a favored seed. On one hand, this seed is likely to cover branches related to validation checks, while those checks are often hard to solve. On the other hand, when this seed initially joins the seed queue, it contains untouched edges. However, as fuzzing proceeds, other seeds may explore these untouched edges, resulting in a seed execution path without untouched edges. Our insight is that if a seed does not contain any untouched edges, there is little sense in prioritizing mutations on it.

In response to the aforementioned challenges, we propose a seed scheduling strategy based on untouched edges extracted from underlying CFG. Unlike traditional seed scheduling strategies that pay more attention to explored regions, our approach gives precedence to seeds that incorporate untouched edges, which are then subjected to fuzzing as a priority. We instrument the target program to collect data on edge coverage and untouched edge information. Subsequently, we revise the untouched edge coverage information for all seeds within the queue. Ultimately, we select an optimal minimal subset of seeds that covers all untouched edges from the seed pool. Furthermore, our scheduling strategy allocates more energy to seeds with more low-frequency untouched edges in the queue. Our insight is that low-frequency untouched edges imply a high probability of being explored, while high-frequency untouched edges are likely to be hard-to-solve edges. Therefore, we enhance the effectiveness of seed scheduling by allocating extra energy to seeds associated with more low-frequency untouched edges, further encouraging in-depth exploration of these areas.

The main contributions of this paper are summarized as follows:

1. We propose a new seed scheduling strategy that efficiently selects seeds based on unexplored regions within program execution paths. This approach prevents wasting resources on ineffective seeds, a common issue in traditional scheduling methods.
2. We applied the new seed scheduling strategy to a new greybox fuzzing tool named UntouchFuzz. To our knowledge, UntouchFuzz is the first fuzzer to utilize unexplored area information as the basis for seed scheduling. The source code is available at https://github.com/bladchan/untouchFuzz.git (accessed on 22 October 2023).
3. We evaluated UntouchFuzz on 12 programs, demonstrating its effectiveness when compared to four AFL-based seed schedulers.

The rest of the paper is organized as follows. Section 2 discusses the background. Section 3 illustrates our motivation with an example. Section 4 shows our design of UntouchFuzz and its technical details. Implementation details are listed in Section 5. Section 6 shows the evaluation results. Section 7 discusses several limitations of our implementation. Section 8 concludes this paper.
2. Background

2.1. Techniques

In this section, we provide the background on coverage-guided greybox fuzzing and focus on the coverage acquisition method and seed scheduling in the classical fuzzer AFL.

2.1.1. Coverage-Guided Greybox Fuzzing

Coverage-guided greybox fuzzing continuously generates test cases by employing coverage feedback loops and preserves seeds that yield new coverage. Specifically, coverage-guided greybox fuzzing includes four main stages [17]:

1. **Seed Scheduling**: Effective seeds are chosen from a pool of seeds based on a scheduling strategy, where effectiveness refers to the fact that new code can be explored more easily by mutating that seed.
2. **Energy Scheduling**: Appropriate mutation counts (energy) are assigned based on the attributes of the chosen effective seeds.
3. **Seed Mutation**: Within the allocated energy, various mutation operations are performed on the selected seeds to generate new test cases.
4. **Seed Selection/saving**: Each generated test case is executed on the target program, and seeds are evaluated based on corresponding coverage information. If a test case enhances the coverage of the target program, it is chosen as a new seed. Through this feedback loop, the coverage of the target program under test continues to increase and it is more likely to generate test cases that trigger new bugs.

2.1.2. Lightweight Instrumentation

The key idea of the coverage-guided greybox fuzzing method lies in coverage acquisition data, as the coverage feedback mechanism propels the entire process of coverage-guided greybox fuzzing forward. AFL, as one of the state-of-the-art coverage-guided fuzzing tools, employs a lightweight instrumentation technique to capture transitions between program basic blocks [18]. AFL assigns a unique random ID to each basic block in CFG. The transition between two basic blocks, i.e., the edge, is defined as in Equation (1).

In particular, \( \text{edge}_i \) represents the ID of the edge, \( \text{prev}_{bb} \) refers to the ID of the previous basic block, and \( \text{cur}_{bb} \) refers to the ID of the basic block where the current transition occurs. Note that \( \text{prev}_{bb} \) is shifted one bit to the right in order to distinguish different transition orders between two basic blocks.

\[
\text{edge}_i = (\text{prev}_{bb} \gg 1) \oplus \text{cur}_{bb} \quad (1)
\]

AFL maintains a default 64 KB bitmap. Each byte in the bitmap is utilized to log the transition count associated with the byte’s index in the bitmap, as illustrated in Figure 1. In this way, AFL can effectively track and count the edges covered by different inputs in the program.
Target Program

if(a>b)

BB1

a>b:

do_something1()

BB2

a<=b:

do_something2()

BB3

① program execute

② calculate index:

(BB1 >> 1) xor BB3

③ update counter

Fuzzer

Counter ... 0B 00 00 01 05 1A ...

Figure 1. Details of AFL edge coverage collection.

2.1.3. Seed Scheduling

AFL employs a genetic algorithm to preserve test cases that cover new edges or hit new edge counts and utilizes a greedy algorithm to generate a favored seed subset from the seed queue, as described in Algorithm 1.

Algorithm 1 Generating the favored seed subset

Input: top rated maintained by AFL
Output: A favored seed set $S_{fav}$

1. $S_{fav} \leftarrow \emptyset$
2. for $i = 0$ to MAP_SIZE do
   3. if $top_{rated}[i]$ and $(temp_v[i] \gg 3) \& (1 \ll (i \& 7))$ then
      4. for $j = (MAP\_SIZE/8 - 1) \rightarrow 0$ do
         5. temp_v $\leftarrow$ temp_v $\& ~ top_{rated}[i].trace_{mini}[j]$
      end
      6. $S_{fav} \leftarrow S_{fav} \cup top_{rated}[i]$
   end
3. return $S_{fav}$

In the initial step, the algorithm maintains an array, $temp_v$, to keep a record of edges currently covered by favored seeds. Then, in line 3, the algorithm iterates through the highest-scoring seed of each edge. Notably, the highest-scoring seeds are dynamically maintained by AFL during the fuzzing process based on criteria such as seed size and execution speed.

Subsequently, the algorithm determines whether the seed has already been covered by other seeds within the favored seed set in lines 4–7. If the edge remains uncovered, we update edge coverage information in $temp_v$. Finally, in line 8, the seed is added to the favored seed set.

Once this favored seed set is obtained, AFL prioritizes these seeds and allocates more energy for mutation. It is important to acknowledge that AFL assumes that if a seed triggers
new edges, fuzzing that seed will likely trigger more edges. However, this assumption has certain limitations when dealing with unexplored regions, as we will discuss in detail in Section 3.

2.2. Related Work

In this section, we discuss the closely related works.

Coverage-guided greybox fuzzing. Coverage-guided greybox fuzzing is one of the most effective techniques for finding vulnerabilities and bugs, garnering significant attention from both academia and industry. Coverage-based greybox fuzzers typically adopt the coverage information to guide different program path explorations.

Since a coverage guidance engine is a key component for the greybox fuzzers, much effort has been devoted to improving their coverage. For example, REDQUEEN [19], GREYONE [20] and PATA [21] employ lightweight taint analysis to penetrate some paths protected by magic bytes comparisons. Driller [22], T-Fuzz [23] and QSYM [3] incorporate symbolic execution engines to delve into deeper program codes. Angora [24] adopts a gradient descent technique to resolve path constraints to break some hard comparisons. MemFuzz [25], MemLock [26], and ovAFLow [27] augment evolutionary fuzzing by additionally leveraging information about memory accesses or memory consumption performed by the target program. CollAFL [28] proposes a coverage-sensitive fuzzing approach to mitigate path collisions. Furthermore, AFLGo [29], Hawkeye [30], Beacon [31], and SelectFuzz [32] utilize alternative metrics for directing fuzzing toward user-specified target sites in the program.

Seed scheduling. In this paper, we focus on improving the seed scheduling component in a fuzzer. With the seed set, seed scheduling is essential for addressing two key issues: (1) which seed to select for the next round and (2) the time budget for the selected seed. In practice, instead of time budget, most fuzzers optimize the number of mutations performed on the selected seeds, i.e., energy scheduling.

AFLFast [12] models path transitions as Markov chain [33], efficiently guiding fuzzing to explore undiscovered path transitions. FairFuzz [13] leverages a targeted mutation strategy to prioritize the exploration of rare branches. EcoFuzz [14] adopts the Variant of the Adversarial Multi-armed Bandit Model (VAMAB) model to prioritize and allocate more energy to the seeds with lower self-transition probabilities. The insight behind it is that the low self-transition probability indicates the high probability of discovering new paths after mutating. Alphuzz [16] models the seed scheduling problem as a Monte Carlo tree search (MCTS) problem. Its key observation is that the relationships among seeds are valuable for seed scheduling. SLIME [15] prioritizes seeds based on reward estimated by a customized upper confidence bound variance-aware (UCB-V) algorithm on different property seed queues, adaptively allocating energy to the seed with different properties.

3. Motivating Example

We use a program’s control flow graph in Figure 2 to illustrate our motivation. The initial seed A is considered to be a quality seed, i.e., the seed is capable of executing the main logic codes of the program. Figure 2a shows the execution path of seed A, with red circles denoting the basic blocks covered by the seed. When the coverage-guided greybox fuzzer mutates seed A, it can easily produce seed B and seed C. Both of these seeds cover edges associated with validation checks at the control flow graph level, i.e., BB1→BB10 and BB2→BB10.

The appearance of these two new edges is interpreted by the fuzzer as an increase in edge coverage, leading to their inclusion in the seed queue. Furthermore, according to the previous description in Section 2.1.3, AFL marks both seed B and seed C as favored seeds during seed scheduling since they cover new edges. However, when attempting to mutate these seeds, they encounter challenges in producing descendant seeds that explore deeper code areas.
We further assume that after mutating seed A, the fuzzer generates not only seed B and seed C but also seed D. Seed D’s execution path is shown in Figure 2d. The difference between seed A and seed D lies in the fact that seed D explores basic block BB6, an unexplored area of seed A’s execution path. In this case, both two seeds are effective for the exploration of basic block BB5. However, if seed E, which covers basic block BB5 is produced later, seed A and seed D become ineffective as they make limited contributions to exploring basic blocks BB8 and BB9. Nevertheless, AFL still regards seed A and seed D as favored because they both cover new edges.

![CFGs of the motivating example](image)

Figure 2. The CFGs of the motivating example.

As discussed in Section 2.1.3, AFL’s seed scheduling algorithm selects favored seeds based on covered edges. However, it lacks tracking information for unexplored areas at the CFG level. AFL’s perspective hinges on the assumption that if a seed covers new edges, it is more likely to explore unexplored regions. This perspective relies on a crucial precondition: the new edges must be adjacent to unexplored areas; otherwise, these edges are likely related to validation checks or have already been explored by other seeds. Particularly in the context of expansive and complex programs, AFL expends substantial fuzzing resources on ineffective seeds, impeding the prioritization of genuinely effective seeds and slowing down the convergence of the fuzzing process. Therefore, this paper addresses the issue in AFL’s seed scheduling algorithm by introducing a new mechanism to track unexplored regions.

### 4. Design of UntouchFuzz

#### 4.1. Overview

Figure 3 shows the overview of UntouchFuzz. In comparison to traditional coverage-guided greybox fuzzers, UntouchFuzz introduces an additional bitmap for tracking untouched edges. The untouched edge instrumentation mechanism updates this bitmap during program execution. Seed scheduling is then performed based on the information from this bitmap, prioritizing the mutations of favored seeds generated by scheduler.

To further explain, UntouchFuzz starts with an initial corpus as a seed set. By using the seed scheduling mechanism, the fuzzer selects a seed from the seed set for mutations. The number of mutations is determined by the energy scheduling mechanism, based on seed attributes. The fuzzer then executes the AFL-instrumented program with mutated test cases. If the program causes a crash, the test case is preserved on the local disk. If it covers new edges, the test case is preserved as a seed and added to the seed queue. Meanwhile, the coverage-increasing seed is provided to the program instrumented for untouched edge tracking, collecting information on untouched edges to guide the next seed scheduling process.
Furthermore, UntouchFuzz allocates more energy to seeds with more low-frequency untouched edges in the seed queue. This allocation aims to encourage these seeds to make more attempts at breaking through these low-frequency untouched edges. In the following sections, we will discuss the methods for collecting information on untouched edges in Section 4.2, introduce the seed scheduling algorithm based on untouched edges in Section 4.3, and outline slight improvements to energy scheduling in Section 4.4.

4.2. Untouched Edges Tracking

As mentioned in Section 2.1.2, AFL employs a lightweight instrumentation technique to track program edge coverage. In essence, AFL’s instrumentation assigns a random ID to each basic block within the target program’s CFG. During program execution, the instrumentation codes calculate the corresponding edge index based on Equation (1) and use this index to update the coverage bitmap.

To ensure minimal impact on program execution speed, AFL utilizes a lightweight XOR operation for computing coverage indices. Similarly, the instrumentation codes responsible for gathering untouched edge information should also be lightweight to minimize disruption to program execution.

We introduce our instrumentation approach with a practical example. Figure 4 provides a snippet of branches within the program’s CFG. The hexadecimal values in green boxes represent random IDs allocated by the AFL instrumentation for each basic block. According to Equation (1), the edge index for the transition from basic block BB1 to BB2 is calculated as \( (0xabcd \gg 1) \oplus 0x1234 = 0x47d2 \), while the edge index for the transition from basic block BB1 to BB3 is calculated as \( (0xabcd \gg 1) \oplus 0x5678 = 0x039e \).

Suppose a particular seed triggers a transition from basic block BB1 to BB2, leaving the transition from BB1 to BB3 unexplored. In this scenario, we label edge \( 0x039e \) as an untouched edge within the execution path of the seed. A straightforward method for capturing untouched edges is to insert instrumentation codes within basic block BB2. Such codes update a byte of untouched edge bitmap by using \( 0x039e \) as a static index pre-allocated during compilation. This approach works efficiently when a basic block has only one predecessor, but confusion arises when a basic block has multiple incoming edges.

Consider the seed’s execution path: BB0 \( \rightarrow \) BB1 \( \rightarrow \) BB2 \( \rightarrow \) BB1. Basic block BB1 has two incoming edges: \( 0xe693 \) from BB0 and \( 0xa2d7 \) from BB2. The corresponding unexplored edges are \( 0xb6a5 \) and \( 0xb2a1 \). Employing the aforementioned instrumentation codes, distinguishing between these two untouched edges becomes a challenging endeavor.
Upon analyzing the above example, it becomes evident that static pre-allocation of untouched edge IDs is impractical when a basic block has multiple predecessors. To address this challenge, we propose a dynamic method for obtaining untouched edge IDs based on the properties of XOR operations. Specifically, we employ a global variable named “__afl_bb_ids” to maintain the XOR value of the IDs of two basic blocks at branch transition point. When a transition between basic blocks occurs, we perform an XOR operation on the ID of the transitioned-to basic block with “__afl_bb_ids” to retrieve the ID of the other unexplored basic block. Furthermore, we define the equation for calculating untouched edge IDs as follows:

\[
edge_{untouch} = (prev_{bb} \gg 1) \oplus (\_afl\_bb\_ids \oplus cur_{bb}) \tag{2}
\]

where, \_afl\_bb\_ids = cur_{bb} \oplus untouched_{bb} \tag{3}

Illustrated with the control flow graph in Figure 4, consider the branch transition point within basic block BB1, which leads to two transitions to basic blocks BB2 and BB3. At BB1, we update the value of “\_afl\_bb\_ids” by performing an XOR operation on the IDs of BB2 and BB3, resulting in \(0x1234 \oplus 0x5678 = 0x444c\). When a transition occurs from basic block BB1 to BB2, we can recover the ID of the unexplored basic block BB3 by using “\_afl\_bb\_ids”: “\_afl\_bb\_ids \oplus ID_{BB2} = 0x444c \oplus 0x1234 = 0x5678”. Subsequently, we calculate the untouched edge ID between BB1 and BB3 using Equation (1): “(0xabcd \gg 1) \oplus 0x5678 = 0x039e”. The calculated value is then utilized to update the bitmap information for the untouched edges.

Algorithm 2 delineates the process of instrumenting untouched edges. Initially, in lines 2–6, the algorithm employs AFL’s native edge coverage-based instrumentation, concurrently capturing the random IDs allocated to each basic block. Subsequently, from lines 8 to 27, the algorithm performs to instrument for untouched edges.

To elucidate further, lines 10–12 of the algorithm determine whether the current “untouch\_inst1” is invoked. “untouch\_inst1” appends instrumentation codes to the beginning of each basic block. The primary function of these codes lies in fetching the value of the global variable “\_afl\_bb\_ids” within the program. It subsequently calculates the ID for the untouched edge per Equation (2) and leverages this ID to update the untouched edge bitmap “\_afl\_untouch\_ptr”.

Figure 4. Example of an AFL-instrumented program’s CFG.
Algorithm 2 Instrumentation for untouched edges

Input: A control flow \( G = (BB, E) \)
Output: A new control flow \( G' = (BB', E') \)

1. \( BB_{IDs} \leftarrow \emptyset, G' \leftarrow \emptyset \)
2. for \( (BB, E) \in G \) do
   // perform AFL's raw instrumentation
   3. \( BB, ID \leftarrow afl_{inst}(BB) \) // ID is assigned by AFL for each basic block
   4. \( BB_{IDs} \leftarrow BB_{IDs} \cup (BB, ID) \)
   end
3. // traverse all basic blocks (our modification)
4. for \( (BB, E) \in G \) do
   // if the current basic block is not the start basic block of G
   5. if \( BB \neq G_.firstBB() \) then
      6. \( BB \leftarrow untouch_{inst1}(BB) \)
   end
   7. \( __afl_bb_ids \leftarrow 0 \)
   8. // traverse all successor basic blocks of the current basic block
   9. for \( BB_{Succ} \in Successors(BB) \) do
      10. for \( (BB_t, BB_{ID}) \in BB_{IDs} \) do
           // find the ID of the successor basic block
           11. if \( BB_{Succ} = BB_t \) then
              12. \( ID \leftarrow BB_{ID} \)
              13. break
           end
      end
   end
   14. \( __afl_bb_ids \leftarrow __afl_bb_ids \oplus ID \)
   end
   15. \( BB \leftarrow untouch_{inst2}(BB, __afl_bb_ids) \)
   16. \( G' \leftarrow G' \cup (BB, E) \)
end
20. return \( G' \)

Moving to lines 15–24, the algorithm traverses the two successor basic blocks of the current basic block, performing XOR operation on the IDs allocated to these two basic blocks. Finally, in line 25, the function “untouch_inst2” is invoked. The primary aim of the code is to update the value of the global variable “\( __afl_bb_ids \)” with the control flow graph, setting it to the outcome of the XOR operation mentioned earlier. At this point, all steps of the untouched edge instrumentation algorithm have been completed.

At the low assembly level, a basic block always has exactly two successor basic blocks. However, when we move up to higher-level compiler intermediate languages, it is not guaranteed that a basic block has always exactly two successor basic blocks, especially in programs that contain Switch statements. Specific solutions to this issue will be provided in Section 5. Additionally, it is worth noting that the instrumentation approach proposed still leads to the problem of edge index collisions, where different edges in the CFG are assigned with the same index value [28].

4.3. Seed Scheduling Based on Untouched Edges

In the preceding Section 4.2, we introduced the method for obtaining untouched edge information. In this section, we delve into seed scheduling based on the collected untouched edges. Similar to AFL, we maintain an array, ‘untouch_top_rated’, with a size of MAP_SIZE to store the most favored seed for each untouched edge. We also employ a greedy algorithm, specifically the minimum covering set algorithm [34], to generate an optimal seed set that contains all currently untouched edges.
Algorithm 3 describes the seed scheduling algorithm based on untouched edges. Initially, in line 1, the algorithm feeds the newly added seed, denoted as ‘s’, to the instrumented program ‘P’ with untouched edges. Subsequently, it acquires the edge coverage bitmap and the untouched edge bitmap of that seed. Following this, in lines 2–27, the algorithm iterates through each index value in the bitmap.

```
Algorithm 3 Seed scheduling based on untouched edges

Input: New seed s, program P' instrumented by Algorithm 1, untouch_top_rated
Output: A favored seed set
1 trace_bits, untouch_bits ← execute(P', s)
2 for i = 0 → MAP_SIZE do
3   if trace_bits[i] then
4     if untouch_bits[i] then
5       untouch_bits[i] = 0 // handle cases when executing in loops
6     end
7     if virgin_untouch[i] then
8       virgin_untouch[i] = 255 // the edge is explored
9     end
10    end
11   if untouch_bits[i] && virgin_untouch[i] ≠ 255 then
12     virgin_untouch[i] = 1 // The edge is not explored by other seeds
13   end
14   if virgin_untouch[i] = 255 then
15     untouch_top_rated[i] ← NULL
16     continue
17   end
18   if untouch_bits[i] ≠ NULL then
19     if untouch_top_rated[i] then
20       fav_factor ← s' execution time × s' file size
21     if fav_factor > untouch_top_rated[i]' execution time × its file size then
22       continue
23     end
24   end
25   untouch_top_rated[i] = s
26 end
27 return coverMinSet(untouch_top_rated)
```

Specifically, in lines 3–10, the algorithm first checks whether the edge has been both marked as a touched edge and an untouched edge in the two bitmaps. If this condition is true, it suggests that the edge is likely in a loop structure. Due to the repeated edges covered in loops, it is possible that edges untouched in a previous iteration of the loop are now covered in the current iteration. To mitigate this effect, the algorithm sets the value of the untouched edge bitmap corresponding to this edge’s index to 0.

The algorithm also maintains a global array called ‘virgin_untouch’. The indices of this array correspond to edge IDs, and the array values indicate the status of the respective untouched edges: 0 denotes an untouched edge that has not been covered by the execution path of any seed, 1 denotes that a seed’s execution path includes this untouched edge, and 255 denotes an untouched edge that has been covered by the execution path of another seed, i.e., it has been “explored”. If the current edge is covered by seed ‘s’, and the corresponding value in the ‘virgin_untouch’ array is not 0, the algorithm updates the value to 255.

Moving on to lines 11–13, if the edge is an untouched edge for the current seed ‘s’ and has not been “explored” by other seeds in history, the algorithm updates the value in the
"virgin_untouch" array to 1. In lines 14–17, if the corresponding "virgin_untouch" value for this edge is 255, indicating that the edge is no longer untouched, the algorithm sets the "untouch_top_rated" value for this edge to NULL. The "untouch_top_rated" maintains the best seed for each untouched edge.

In lines 18–26 of the algorithm, the current seed \( s \) is compared with the metrics of the best seed for this untouched edge. We continue to use AFL’s default metrics, which is the seed’s execution speed multiplied by its file size. Then, at line 28, the algorithm invokes the \( \text{coverMinSet}() \) function, which generates a minimal seed set containing all untouched edges found, following the logic described in Algorithm 1.

In the end, the algorithm outputs the optimal seed set based on untouched edges. UntouchFuzz prioritizes testing seeds from this selected seed set.

### 4.4. Energy Scheduling Optimization

The goal of energy scheduling is to allocate energy efficiently to the chosen seeds for optimal mutations. It is essential to strike the right balance in energy allocation. Allocating excessive energy can lead to a significant waste of fuzzing resources on a single seed. Conversely, insufficient energy allocation may underutilize a seed’s potential to explore new paths, as discussed in reference [14].

In this paper, we obtain the number of seeds in the seed set for each untouched edge included. Following this, we sort the number of seeds for these untouched edges in ascending order and select the top \( \beta \% \) (40% is the default in this paper) as rare untouched edges. Then, we calculate the difference between the number of current seed’s rare untouched edge \( \text{rare}_s[i] \) and the maximum number of seed \( \text{max}_s \) among all rare untouched edges then calculate the average distance value \( \text{dist}_s \) based on Equation (4).

\[
\text{dist}_s = \frac{\sum (\text{max}_s - \text{rare}_s[i])}{n} \tag{4}
\]

Assuming the original energy allocated to the seed was \( p \), it is now assigned a new energy of \( (1 + \alpha \times \text{dist}_s) \times p \), where \( \alpha \) is the default value of 0.3. In our insight, if a certain untouched edge appears frequently in the seed set, it implies a high probability that the selected seed contains this untouched edge and suggests that this particular untouched edge is likely to be difficult to explore. Therefore, if a seed includes many high-frequency untouched edges, it should be allocated less energy. Conversely, if the seed contains numerous low-frequency untouched edges, it should be allocated more energy.

To elaborate on our insight, Figure 5 illustrates it through an example. Assuming that the current phase has fuzzed for a while and the seed set contains four seeds, with two global untouched edges: \( \text{BB1} \rightarrow \text{BB10} \) and \( \text{BB5} \rightarrow \text{BB8} \). According to the seed scheduling algorithm outlined in Section 4.3, seed B is identified as a favored seed due to the presence of untouched edge \( \text{BB1} \rightarrow \text{BB10} \) in its execution path and superior seed attribute. Similarly, seed D is designated as a favored seed because of the inclusion of the new untouched edge \( \text{BB5} \rightarrow \text{BB8} \) in its execution path.

Subsequently, the fuzzing process prioritizes the mutation of seed B and seed D. As previously mentioned, the untouched edge \( \text{BB1} \rightarrow \text{BB10} \) appears in the execution paths of all four seeds, indicating it is not a rare untouched edge. In contrast, \( \text{BB5} \rightarrow \text{BB8} \) is considered as a rare untouched edge since it appears only in the execution path of seed D. Consequently, we can make a reasonable conjecture that the untouched edge \( \text{BB1} \rightarrow \text{BB10} \) is likely associated with a hard-to-solve constraint, whereas \( \text{BB5} \rightarrow \text{BB8} \) is likely to represent a more manageable constraint. To increase the likelihood of covering edge \( \text{BB5} \rightarrow \text{BB8} \), we allocate more energy to the mutation of seed D based on the previously outlined energy allocation mechanism. It is essential to note that the provided example is simplified for explanatory purposes, while the real-world program will be complex.

However, in AFL, when new seeds are discovered, the fuzzer doubles the energy allocated to the seeds. Hence, we do not intentionally reduce the energy but instead
provide seeds with a higher initial energy value to the seeds, aiming to fully unleash the potential of seeds to discover new paths.

Figure 5. Example of our insight.

5. Implementation

We implemented a prototype of UntouchFuzz on the top of AFL 2.57b, comprising two key components: instrumentation based on untouched edges and the main fuzzing loop. Next, we discuss a few important implementation details.

For the instrumentation, we employed the LLVM framework [35] to instrument the target program’s codes, collecting information about untouched edges. However, LLVM IR contains SwitchInst instructions, which can lead to situations where basic blocks with SwitchInst instructions have multiple successor basic blocks. To address this, we utilized a pass available in the AFL++ [6] instrumentation tools that splits switch statements. By applying this pass, all SwitchInst instructions in the target program’s LLVM IR are transformed into if…else… structures, converting basic blocks that originally had multiple successor basic blocks into those with only two successor basic blocks. This enables us to effectively implement the instrumentation method described in Section 4.2.

Before entering the main fuzzing loop, we launched an instrumented program using another fork server for untouched edge instrumentation in UntouchFuzz. Here, we considered that instrumentation due to untouched edges might impact the program’s execution speed. Therefore, we only run the untouched edge-instrumented version of the program when seeds are added to the queue. In other scenarios, we run the AFL’s native instrumented version of the program. However, it is important to note that to maintain instrumentation consistency, we applied the aforementioned switch statement, splitting pass to the AFL’s native instrumentation.

Regarding the fuzzing main loop, we introduced an update_untouch_score() function to maintain the best seed for each untouched edge. Additionally, we made modifications to the cull_queue() and fuzz_one() functions in AFL to implement seed scheduling and energy allocation. Other logic in the fuzzing main loop remained unchanged.

6. Evaluation

In this section, we evaluated the effectiveness of UntouchFuzz and answer the following questions:

- RQ1: How effective is UntouchFuzz at improving coverage faster when compared with other seed scheduling strategies?
- RQ2: Can UntouchFuzz discover more unique crashes with respect to other seed scheduling strategies?
- RQ3: How does the seed scheduling based on untouched edges perform in other fuzzers?
- RQ4: Can UntouchFuzz detect new vulnerabilities in real-world programs?
6.1. Experiment Settings

(1) Baseline Seed Scheduling Strategies: Starting from the minimum coverage set, rare paths, new paths, and same-prefix coverage, the seed scheduling method proposed in this paper was compared with native AFL [9], AFLFast [12], EcoFuzz [14], and Alphuzz [16] seed scheduling strategies. It is important to note that these five tools differ only in their seed scheduling mechanisms while all other components remain the same. FairFuzz [13] was not compared due to its custom mutator, which aims to obtain mutation byte masks. When fuzzing large-scale seeds, the custom mutator might lead to starvation in subsequent seeds. Conversely, if seeds are small, the additional coverage gained from deterministic mutation might be unfair compared to fuzzers without deterministic mutation. Previous work [36] removed this custom mutation phase in experiments, but this deviated from FairFuzz’s original intent. Therefore, we do not select FairFuzz for comparison.

Additionally, the multi-property queue seed scheduling method SLIME [15] is not compared due to the additional instrumentation for the target program. This instrumentation affects program execution speed and differs significantly from the instrumentation used by the aforementioned tools. The tool we implemented can share the same instrumentation program with these tools, ensuring no experimental differences in comparison.

(2) Benchmark Programs: We selected 12 real-world binary programs for testing based on their popularity, testing frequency, and diversity of categories. As shown in Table 1, these 12 binary programs include popular binary utilities (such as readelf), image parsing and processing libraries (such as libjpeg-turbo, exiv2), audio parsing tools (such as mp3gain), document processing libraries (such as xpdf, libxml2), and network packet parsing tools (such as tcpdump). Since certain vulnerabilities (i.e., buffer overflows) do not affect the program execution, we apply Address Sanitizer (ASAN) [37] to capture memory errors.

Table 1. Twelve real-world programs for evaluation.

<table>
<thead>
<tr>
<th>Program</th>
<th>Library and Version</th>
<th>Input Type</th>
<th>Commands</th>
</tr>
</thead>
<tbody>
<tr>
<td>djpeg</td>
<td>libjpeg-turbo-2.1.91</td>
<td>jpg</td>
<td>@@</td>
</tr>
<tr>
<td>exiv2</td>
<td>exiv2-0.26</td>
<td>jpg</td>
<td>@@ /dev/null</td>
</tr>
<tr>
<td>pdftotext</td>
<td>xpdf-4.0.0</td>
<td>pdf</td>
<td>@@</td>
</tr>
<tr>
<td>tcmdump</td>
<td>tcmdump-4.8.1</td>
<td>bin</td>
<td>@@ /dev/null</td>
</tr>
<tr>
<td>mp3gain</td>
<td>mp3gain-1.5.2</td>
<td>mp3</td>
<td>@@</td>
</tr>
<tr>
<td>mp42aac</td>
<td>Bento4-1.5.1-628</td>
<td>mp4</td>
<td>@@ /dev/null</td>
</tr>
<tr>
<td>tiffcp</td>
<td>libtiff-3.9.7</td>
<td>tiff</td>
<td>-i-E1-H10-V10-S84-R270 @@/output.tif</td>
</tr>
<tr>
<td>readelf</td>
<td>binutils-2.28</td>
<td>elf</td>
<td>-a @@</td>
</tr>
<tr>
<td>nm-new</td>
<td>binutils-2.28</td>
<td>elf</td>
<td>-A -a -l -S -s -special-syms --synthetic --with-symbol-versions -D @@</td>
</tr>
<tr>
<td>xmlmlint</td>
<td>libxml-2.98</td>
<td>xml</td>
<td>@@</td>
</tr>
<tr>
<td>bsdtar</td>
<td>libarchive-3.2.0</td>
<td>tar</td>
<td>-xf @@ /dev/null</td>
</tr>
<tr>
<td>mujs</td>
<td>SQLite-3.8.9</td>
<td>text(js)</td>
<td>@@</td>
</tr>
</tbody>
</table>

(3) Initial Corpus: The initial seed corpus used comes from datasets provided by Mopt [38] and uniFuzz [39], with some contributions from the open-source community.

(4) Experimental Environment: The experiments were conducted on a server with a 56-core Intel Xeon CPU, 128 GB of memory, and running Ubuntu 22.04. Deterministic mutations were disabled as it is less effective compared to AFL’s havoc mutation strategy [40]. To reduce the impact of randomness, we ran each benchmark program for 24 h, repeating the process 10 times and taking the arithmetic mean as the final result [41].

(5) Experimental Metrics: We evaluated the proposed method against four fuzzers with different seed scheduling strategies, considering edge coverage, edge coverage over time, and the number of unique crashes. Additionally, the proposed guided mechanism for untouched edges was transplanted into MOpt to assess its performance across different
fuzzers. MOpt was chosen due to its focus on optimizing mutation operators while keeping the seed scheduling mechanism unaltered.

### 6.2. RQ1: Code Coverage Improving

Code coverage is a crucial metric for evaluating the performance of fuzzing techniques [17]. In general, the more code a fuzzer can cover in the target program, the higher the probability of discovering hidden vulnerabilities. As explained in Section 2.1.2, AFL [9] employs a 64KB bitmap to collect coverage information, with each byte in the bitmap representing the number of hits for a particular edge ID. AFL maps program branches to the bitmap by using a hash function. If a branch is explored, the byte at the index corresponding to its edge ID in the bitmap is updated. AFL maintains a simplified bitmap in real time and stores it on local disk, which allows us to assess code coverage based on this simplified bitmap.

Table 2 presents a comparison of UntouchFuzz with four other state-of-the-art fuzzers in terms of edge coverage. The data in the table represent the arithmetic average edge coverage across ten fuzzing tests. The results in Table 2 demonstrate that UntouchFuzz outperforms EcoFuzz and Alphuzz in edge coverage and slightly surpasses AFL and AFLFast. UntouchFuzz achieves better coverage than the other four baseline seed scheduling strategies in 11 out of the 12 programs tested (all except for mujs). Overall, the proposed method is effective in improving coverage. While the improvement percentages on AFL and AFLFast (2.16% and 3.31%) are relatively modest, these results are consistent with findings from previous research [40], which suggests that differences among fuzzers are minimized when using the single havoc strategy. Nonetheless, our seed scheduling strategy still has an impact on the direction of fuzzing evolution by concentrating mutation energy on more effective seeds, thus enhancing overall program coverage.

**Table 2. Arithmetic mean edge coverage comparison.**

<table>
<thead>
<tr>
<th>Fuzzer</th>
<th>Default</th>
<th>RarePath</th>
<th>NewPath</th>
<th>SamePrefix</th>
<th>UntouchedEdge</th>
</tr>
</thead>
<tbody>
<tr>
<td>djpeg</td>
<td>3998</td>
<td>3850</td>
<td>3646</td>
<td>3613</td>
<td>4101</td>
</tr>
<tr>
<td>exiv2</td>
<td>12,152</td>
<td>12,104</td>
<td>11,310</td>
<td>11,958</td>
<td>12,304</td>
</tr>
<tr>
<td>pdftotext</td>
<td>15,160</td>
<td>15,120</td>
<td>11,408</td>
<td>14,510</td>
<td>15,280</td>
</tr>
<tr>
<td>tcpdump</td>
<td>18,208</td>
<td>17,252</td>
<td>16,134</td>
<td>17,640</td>
<td>18,556</td>
</tr>
<tr>
<td>mp3gain</td>
<td>1375</td>
<td>1376</td>
<td>1364</td>
<td>1370</td>
<td>1382</td>
</tr>
<tr>
<td>mp42aac</td>
<td>3263</td>
<td>3230</td>
<td>3178</td>
<td>3197</td>
<td>3263</td>
</tr>
<tr>
<td>tiffcp</td>
<td>5841</td>
<td>5801</td>
<td>5405</td>
<td>5692</td>
<td>6659</td>
</tr>
<tr>
<td>readelf</td>
<td>10,336</td>
<td>10,503</td>
<td>9100</td>
<td>10,029</td>
<td>10,647</td>
</tr>
<tr>
<td>nm-new</td>
<td>5455</td>
<td>5463</td>
<td>5530</td>
<td>5185</td>
<td>5601</td>
</tr>
<tr>
<td>xmllint</td>
<td>10,406</td>
<td>10,332</td>
<td>10,121</td>
<td>10,172</td>
<td>10,470</td>
</tr>
<tr>
<td>bsdtar</td>
<td>5250</td>
<td>5227</td>
<td>5109</td>
<td>5169</td>
<td>5318</td>
</tr>
<tr>
<td>mujs</td>
<td>9170</td>
<td>9243</td>
<td>8892</td>
<td>9084</td>
<td>9210</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>100,614</td>
<td>99,501</td>
<td>91,197</td>
<td>97,619</td>
<td>102,791</td>
</tr>
</tbody>
</table>

Figure 6 shows the evolution of edge coverage over time, with samples taken every hour. Evidently, on the seven target programs, exiv2, pdftotext, tcpdump, tiffcp, readelf, nm-new, and bsdtar, UntouchFuzz has a significantly higher edge coverage growth rate than the other four baseline seed scheduling strategies. However, for the remaining five target programs, the fuzzers converge quickly due to their smaller scale and lower complexity, negating the advantage demonstrated by UntouchFuzz.
6.3. RQ2: Unique Crashes

To further validate the effectiveness of the proposed method in vulnerability discovery, we conducted a statistical analysis of unique crashes. Table 3 presents a comparison of UntouchFuzz with four baseline fuzzers on the number of unique crashes, where the data in the table represent the total number of unique crashes discovered over ten rounds of testing. It is worth noting that, as in our experiments, we used the latest version of the djpeg program and did not find any valid crashes, so its results are not listed in Table 3.

As shown in the experimental results in Table 3, UntouchFuzz outperforms the other four baseline seed scheduling strategies in total unique crash discoveries, discovering the highest number of unique crashes on six of the programs under test. However, on the remaining five target programs, UntouchFuzz does not achieve the best results, but the difference between it and the best-performing approach is not significant. We attribute this to the randomness of mutation and differences in fuzzing evolution.

It is essential to note that the experimental results in Table 3 are directly taken from the unique crash metric of AFL-based fuzzers. However, the number of unique crashes may not accurately reflect the actual number of unique vulnerabilities because there is often a many-to-one relationship between them, which means that multiple crashes may correspond to a single vulnerability. For example, suppose crash one’s triggering path is A → B → D, and crash two’s triggering path is A → C → D, and both crashes occur at the same location in basic block D. From AFL’s perspective, since the triggering paths of these two crashes are different, they are both considered unique crashes. However, from a root cause analysis perspective, both crashes are due to codes in basic block D, and thus, these two crashes should be categorized as the same vulnerability. Moreover, the differences in these two crash paths are likely related to changes in the input, where bytes in the input can affect the execution of subsequent basic blocks following basic block A. To obtain a more accurate count of unique vulnerabilities, we conducted deduplication of unique crashes based on the function call stack information provided by ASAN, selecting the top three functions and removing duplicate crashes. Table 4 presents the results after crash deduplication.
The data in Table 4 reveal that various seed scheduling strategies exhibit distinct performances across different programs, with UntouchFuzz achieving the highest number of unique vulnerabilities in six programs. Overall, UntouchFuzz outperforms the other four baseline seed scheduling strategies based on the total number of discovered vulnerabilities. Furthermore, compared to EcoFuzz and Alphuzz, UntouchFuzz demonstrates a more stable performance across the programs.

6.4. RQ3: Scalability

To assess the scalability of the proposed approach, we integrated our method into the MOpt fuzzer, naming the modified tool “MOpt-u.” Table 5 provides a comparison of edge coverage between UntouchFuzz, AFL, MOpt-u, and the original MOpt fuzzer. The results in Table 5 demonstrate that UntouchFuzz and MOpt-u outperform the original, unmodified fuzzers in terms of edge coverage across all 12 benchmark programs. This further substantiates the capability of the untouched edge-guided mechanism to enhance the performance of the original fuzzers.
Table 5. Comparison between AFL, MOpt and UntouchFuzz, MOpt-u.

<table>
<thead>
<tr>
<th>Fuzzer</th>
<th>AFL</th>
<th>UntouchFuzz</th>
<th>Mopt</th>
<th>Mopt-u</th>
</tr>
</thead>
<tbody>
<tr>
<td>djpeg</td>
<td>3998</td>
<td>4101</td>
<td>4073</td>
<td>4397</td>
</tr>
<tr>
<td>exiv2</td>
<td>12,152</td>
<td>12,304</td>
<td>12,404</td>
<td>12,480</td>
</tr>
<tr>
<td>pdftotext</td>
<td>15,160</td>
<td>15,280</td>
<td>15,322</td>
<td>15,341</td>
</tr>
<tr>
<td>tcpdump</td>
<td>18,208</td>
<td>18,356</td>
<td>18,467</td>
<td>18,734</td>
</tr>
<tr>
<td>mp3gain</td>
<td>1375</td>
<td>1382</td>
<td>1378</td>
<td>1380</td>
</tr>
<tr>
<td>mp42aac</td>
<td>3263</td>
<td>3263</td>
<td>3352</td>
<td>3409</td>
</tr>
<tr>
<td>tifffp</td>
<td>5841</td>
<td>6659</td>
<td>5949</td>
<td>6132</td>
</tr>
<tr>
<td>readelf</td>
<td>10,336</td>
<td>10,647</td>
<td>11,128</td>
<td>11,252</td>
</tr>
<tr>
<td>nmm-new</td>
<td>5455</td>
<td>5601</td>
<td>5573</td>
<td>5674</td>
</tr>
<tr>
<td>xmlint</td>
<td>10,406</td>
<td>10,470</td>
<td>10,245</td>
<td>10,408</td>
</tr>
<tr>
<td>bsd.tar</td>
<td>5250</td>
<td>5318</td>
<td>5270</td>
<td>5273</td>
</tr>
<tr>
<td>mujs</td>
<td>9170</td>
<td>9210</td>
<td>9121</td>
<td>9157</td>
</tr>
<tr>
<td>Total</td>
<td>100,614</td>
<td>102,791</td>
<td>102,282</td>
<td>103,637</td>
</tr>
<tr>
<td></td>
<td>(−2.16%)</td>
<td>(−1.31%)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

6.5. RQ4: New Vulnerabilities

We used UntouchFuzz to find new vulnerabilities in open-source projects on GitHub. We reported these vulnerabilities to the respective projects. The details are in Table 6, confirming the effectiveness of UntouchFuzz in real-world scenarios.

Table 6. New vulnerabilities found by UntouchFuzz.

<table>
<thead>
<tr>
<th>Project</th>
<th>Version</th>
<th>CVE/Issue</th>
<th>Type</th>
<th>Fixed</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIEF</td>
<td>v0.12.1</td>
<td>CVE-2022-40922</td>
<td>segmentation fault</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>LIEF</td>
<td>v0.12.1</td>
<td>CVE-2022-40923</td>
<td>segmentation fault</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>LIEF</td>
<td>v0.12.1</td>
<td>CVE-2022-43171</td>
<td>heap buffer overflow</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>LIEF</td>
<td>v0.12.1</td>
<td>CVE-2022-43172</td>
<td>segmentation fault</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>LIEF</td>
<td>v0.12.1</td>
<td>github-issue-785</td>
<td>allocator oom</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>PcapPlusPlus</td>
<td>v22.11</td>
<td>CVE-2023-31991</td>
<td>heap buffer overflow</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>libfyaml</td>
<td>v0.7.12</td>
<td>CVE-2023-31992</td>
<td>use after free</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>libfyaml</td>
<td>v0.7.12</td>
<td>CVE-2023-31993</td>
<td>stack buffer overflow</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>libfyaml</td>
<td>v0.7.12</td>
<td>github-issue-56</td>
<td>stack-buffer-overflow</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>sxmlc</td>
<td>v4.5.2</td>
<td>github-issue-24</td>
<td>segmentation fault</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>sxmlc</td>
<td>v4.5.2</td>
<td>github-issue-25</td>
<td>segmentation fault</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>configor</td>
<td>v0.9.18</td>
<td>github-issue-97</td>
<td>infinite loop</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>tom11</td>
<td>v3.7.1</td>
<td>github-issue-199</td>
<td>heap buffer overflow</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

7. Discussion

In this section, we discuss several limitations of our current implementation:

1. Our method is implemented on the top of AFL and not on AFL++. The choice to not implement it on AFL++ was due to the fact that AFL++ already integrates the seed scheduling mechanism from AFLFast and other advanced technologies. Porting the untouched edge guidance mechanism into AFL++ would have been complex. We plan to implement our approach to AFL++ in future work. Additionally, it is important to note that our method may not apply to base fuzzers outside of AFL, such as libFuzzer [42] or honggfuzz [1].

2. Coverage bitmap collisions are a common issue in AFL-based fuzzers. AFL-based fuzzers typically use a fixed-size 64KB bitmap to collect coverage information, which can be adjusted via configuration. The fixed bitmap size might result in different edges being assigned the same edge ID, causing coverage bitmap collisions. Prior research attempts have aimed to optimize this issue by modifying instrumentation [28,43,44]. However, with the increase in the size of the target programs, expanding the bitmap to solve collision issues might not lead to significant performance improvements.
(3) We conducted fuzzing tests on 12 mainstream benchmark programs, running each program 10 times for 24 h per test. The experimental results demonstrate that, under the given initial corpus conditions, our tool effectively selects better seeds for fuzzing and guides the fuzzer toward maximizing program coverage. However, differences between the initial corpus and the target program can impact the performance of coverage-guided greybox fuzzers [10,15,45], causing variations in the evolutionary process and resulting in different outcomes.

8. Conclusions

In this paper, we concluded that the existing seed scheduling methods neglect to focus on the unexplored regions within the program’s control flow graph. In response to this issue, we presented a greybox fuzzer guided by untouched edges, UntouchFuzz. We developed a lightweight instrumentation technique to track untouched edges. Furthermore, we designed a seed scheduling strategy based on untouched edges inspired by a minimal coverage sets algorithm. The strategy prioritizes seeds that include all untouched edges. Additionally, we made minor adjustments to the energy scheduler to align with the new seed scheduling method. In evaluation, UntouchFuzz outperformed the other fuzzers on code coverage and the number of vulnerabilities, further proving the untouched guidance mechanism proposed in this paper. To foster future research in this area, we have made our fuzzer open source. Further, future research could combine symbolic execution techniques with the untouched guidance mechanism for better fuzzing results. We plan to implement our mechanism into AFL++ and investigate the influence of bitmap collisions in our mechanism.

Author Contributions: Conceptualization, P.J.; data curation, C.X.; formal analysis, G.Y.; funding acquisition, P.J.; investigation, H.K.; methodology, C.X.; software, C.X.; supervision, P.Y. and C.H.; validation, X.H.; visualization, C.X.; writing—original draft, C.X.; writing—review and editing, C.X. and P.J. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by National Key R&D projects of China OF FUNDER grant number 2021YFB3101803. This research was supported in part by the National Natural Science Foundation of China under Grant U2133208 and the Sichuan Youth Science and Technology Innovation Team under Grant 2022JDTD0014.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: The source code of fuzzer proposed in this research is available at https://github.com/bladchan/untouchFuzz.git (accessed on 22 October 2023).

Conflicts of Interest: The authors declare no conflict of interest.

References


Disclaimer/Publisher’s Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.