Research Article

Journal of Computer Security
2025, Vol. 33(5) 334–359

DOI: 10.1177/0926227X251343867
journals.sagepub.com/home/jcu

HuntFUZZ: Enhancing error handling testing through clustering based fuzzing

Jin Wei[1,2], Ping Chen[2,3], Jun Dai[4], Xiaoyan Sun[4], Zhihao Zhang[4], Chang Xu[1] and Yi Wang[2]

Abstract

Testing a program's capability to effectively handle errors is a significant challenge, given that program errors are relatively uncommon. To address this, software fault injection (SFI)-based fuzzing combines SFI with traditional fuzzing to inject faults and trigger errors, enabling the testing of (error handling) code. However, current SFI-based fuzzing approaches have overlooked the correlation between paths housing error points. In fact, the execution paths of error points often share common paths. As a result, fuzzers usually generate test cases repeatedly to explore these common paths. This practice can compromise the efficiency of the fuzzer(s). To address this issue, this paper introduces HuntFUZZ, a novel SFI-based fuzzing framework designed to minimize redundant exploration of error points with correlated paths. HuntFUZZ achieves this by clustering these correlated error points and using concolic execution to resolve the path constraints necessary for approaching or reaching these clusters. This approach provides the fuzzer with optimized test cases, allowing it to efficiently explore error points within the cluster while minimizing redundancy. We evaluate HuntFUZZ on a diverse set of 42 applications, and HuntFUZZ successfully reveals 162 known bugs, with 62 of them being related to error handling. Additionally, due to its efficient error point detection method, HuntFUZZ discovers seven unique zero-day bugs, which are all missed by existing fuzzers. Furthermore, we compare HuntFUZZ with four existing fuzzing approaches, including AFL, AFL++, AFLGo, and EH-FUZZ. Our evaluation confirms that HuntFUZZ can cover a broader range of error points, and it exhibits better performance in terms of bug-finding speed.

Keywords

Error handling, software fault injection, fuzzing, concolic execution, hybrid fuzzing

Received: 15 April 2024; accepted: 22 April 2025

1 Introduction

Real-world programs require robust error handling to manage various potential issues that may arise during execution. Exceptional situations, such as invalid inputs, memory shortages, integer overflows, and network connection failures, can occur due to specific conditions. These situations are generally referred to as errors, and the code responsible for managing them is known as error handling code. However, error handling in programs is often flawed or even missing altogether. Testing whether a program can handle errors properly is quite challenging because error handling code is rarely executed in typical program workflows, simply due to the infrequency of errors occurring in practice.[1–3] Additionally, testing for error handling may be inadequate because it is inherently difficult to trigger these rare error conditions and reach the corresponding error handling points during standard testing processes.[1–8] Insufficient error handling can lead to severe security consequences.[9–12] Examples of such vulnerabilities include CVE-2019-7846, which leads to information

[1]School of Computer Science, Fudan University, Shanghai, China
[2]Institute of BigsData, Fudan University, Shanghai, China
[3]Purple Mountain Laboratories, Nanjing, China
[4]Worcester Polytechnic Institute, Massachusetts, USA

Corresponding author:
Ping Chen, Institute of BigData, Fudan University, Shanghai, China; Purple Mountain Laboratories, Nanjing, China.
Email: pchen@fudan.edu.cn

disclosure[13]; CVE-2019-2240, which causes abnormal program functionality[14]; and CVE-2019-1750 and CVE-2019-1785, both of which can result in denial of service.[15,16] Therefore, testing whether a program can effectively handle errors is crucial for mitigating potential security risks.

To enhance the testing of error handling, recent approaches involve the utilization of software fault injection (SFI)-based fuzzing approaches.[17–21] These methods combine SFI[22] and fuzzing technologies.[23–39] Specifically, SFI involves injecting faults or errors into the tested program, so that the program can be executed to test whether it can effectively handle the injected faults or errors.[19] The code locations where errors are injected are referred to as error points. Fuzzing is then employed to generate program inputs as test cases to cover these error points. Thus, SFI-based fuzzing approaches require both the generation of error sequences (which are ordered sets of error points, represented by 0 or 1) to indicate where to insert errors at potential sites[21] and the generation of program inputs to reach the error points. Existing SFI-based fuzzing approaches primarily focus on optimizing the generation of error sequences. For instance, FIFUZZ[19] employs context-sensitive SFI to cover error points in different calling contexts. iFIZZ[20] adopts a state-aware SFI approach, defining state as the runtime context of an error site, to reduce redundant fault scenarios. EH-FUZZ[21] utilizes error coverage—a metric of a fuzzer's capability to test the number of injection scenarios for error points, which will be explained in detail in Section 5.2.2—to direct error injection. This helps avoid exploration of duplicate error scenarios and attempts to detect more diverse error circumstances.

```
1  // Patch for function memalign
2  --- a/malloc/malloc.c
3  +++ b/malloc/malloc.c
4  @@ -3015,6 +3015,13 @@
5  __libc_memalign(size_t alignment, size_t bytes)
6  {  ...
7     if (alignment <  MINSIZE) alignment = MINSIZE;
8
9  +   /* Check for overflow. */
10 +  if (bytes > SIZE_MAX - alignment - MINSIZE)
11 +   {
12 +     __set_errno (ENOMEM);
13 +     return 0;
14 +    }
15    arena_get(ar_ptr, bytes + alignment + MINSIZE);
16    if(!ar_ptr)
17       return 0;
18 }
```

Code 1 Patch for function `memalign` in `malloc.c` from GNU C Library 2.18 or earlier.

Despite substantial efforts dedicated to generating error sequences, we find some insights that have not been considered in existing research on generating program inputs. Specifically, we observe that many error points exhibit significant correlation because they share (longest) common paths. However, existing fuzzers do not take this factor into account when generating program inputs, resulting in fuzzers repeatedly generating program inputs to cover these common paths. For example, the CVE-2013-4332 vulnerability[40] indicates multiple integer overflows in malloc.c in GNU C Library version 2.18 and earlier. These vulnerabilities allow an attacker to cause integer overflow by manipulating the variable bytes in function memalign, valloc, and pvalloc, leading to a denial of service (heap corruption). The vulnerability arises due to the lack of checks on the variable bytes in the program and the absence of capability of error handling when the value of bytes exceeds a certain range. As shown in Code 1, lines 9–14 demonstrate the patch addressing the overflow vulnerability in the memalign function. Specifically, a check is introduced in this patch to examine the variable bytes. If the value of this variable exceeds SIZE_MAX - alignment - MINSIZE, an error message is thrown. Without this necessary check, integer overflow may happen. Similar overflow vulnerabilities also exist in the functions valloc and pvalloc, all due to the lack of checks on the variable bytes. Furthermore, memalign, valloc, and pvalloc reside within the same switch case structure and share a common execution path, specifically $\cdots \rightarrow$ allocate_thread $\rightarrow$ allocate $\rightarrow$ allocate_1 $\rightarrow$ switch case (as shown in Code 2). This suggests that if a fuzzer separately solves the error points for functions memalign, valloc, and pvalloc, it may redundantly solve the common execution path, thereby diminishing the efficiency of the fuzzer. We elaborate on this aspect in Section 2.2.

To enhance the capability of the SFI-based fuzzer in exploring error points, this paper proposes an optimization strategy that incorporates concolic execution to expedite the process of reaching related error points that share common paths. To implement this strategy, our approach begins by clustering error points that exhibit common paths, ensuring that within each cluster, the distance between all error points and their common parent node is less than a specified threshold. Next, we evaluate the significance of each cluster by calculating its weight, which is determined by both the number of injected error

points within the cluster and the distance of the cluster from the current path. This evaluation allows us to prioritize which cluster's error points to explore, focusing on those most likely to lead to important and diverse findings. Subsequently, we derive constraints for the paths that enable approaching or reaching the prioritized cluster. By providing the fuzzer with test cases that meet these constraints, it can efficiently explore related error points with minimal redundancy.

These strategies are integrated into a fuzzing framework named HuntFUZZ. We conducted a thorough experimental evaluation to validate its effectiveness and performance. The experimental results demonstrated that HuntFUZZ identified known error-handling bugs, and it also discovered seven zero-day bugs. Additionally, we compared HuntFUZZ with several state-of-the-art fuzzing methods, showing that it exhibits stronger error point testing capabilities and broader error point coverage.

In conclusion, this paper makes the following contributions:

- We present HuntFUZZ, a novel SFI-based fuzzing framework designed to improve the efficiency of error point detection through clustering techniques. Additionally, we propose an optimization algorithm that incorporates concolic execution to effectively resolve input constraints for error points within each cluster. This approach facilitates more targeted testing by guiding the fuzzer toward specific clusters of error points, while avoiding redundant exploration of error points that share common paths. As a result, HuntFUZZ significantly enhances the fuzzer's effectiveness in error point testing. Our findings also demonstrate HuntFUZZ's capability in effectively exploring deep-state error points—defined in this paper as error points with a depth exceeding 500 in the control flow graph (CFG), as discussed in Section 5.2.1. This is attributed to the integration of concolic execution, which aids in testing some deep-state error points dependent on very intricate and specific constraints.
- We evaluate HuntFUZZ across a diverse spectrum of 42 applications, including two datasets (Unibench[41] and programs previously tested by EH-FUZZ[21]) as well as nine additional non-benchmark programs. HuntFUZZ successfully discovered a total of 162 known bugs, including 62 error-handling bugs. Additionally, HuntFUZZ uncovered seven zero-day bugs. We compare it with four established fuzzing approaches (AFL[34], AFL++[42], AFLGo[43], and EH-FUZZ[21]). The results affirm that HuntFUZZ can discover more error-handling bugs and achieve accelerated and superior coverage of error points. Notably, compared to the contemporary SFI-based fuzzing method (i.e. EH-FUZZ), HuntFUZZ exhibits a remarkable 38.9% increase in error coverage.
- We have made HuntFUZZ available, and the source code can be accessed at https://github.com/weijinjinnihao/HuntFUZZ.

2 Background and key insights

2.1 Background

2.1.1 SFI-based fuzzing for error-handing test. Although errors in the program are not frequent, failure to handle errors properly can lead to serious security vulnerabilities, posing a significant threat to the normal operation of the system. Examples of such threats include denial of service, information disclosure, local privilege escalation, and other critical impacts.[21] While some traditional fuzzers[23,39] are adept at discovering some errors by rapidly generating program inputs, these input-driven fuzzing approaches often fall short in detecting input-independent errors, because these types of errors typically stem from exceptional events that sporadically occur, such as insufficient memory or network connection failures. Thus, traditional fuzzers prove ineffective in handling errors.

To overcome the limitations of traditional fuzzers, researchers introduce SFI[8,22,44–8] into traditional fuzzing to trigger input-independent errors and force the execution of error paths. Specifically, SFI introduces faults or errors into the tested program and then runs the program to test whether it can effectively handle the injected faults or errors.[19] The code locations where errors are injected are referred to as error points. SFI-based fuzzing typically begins by conducting a static analysis of the source code of the tested program to identify error points. Subsequently, the fuzzer mutates error sequences, indicating whether the error points can execute normally or fail, based on the calling context of error points.[21] Then, SFI-based fuzzing approaches follow the traditional fuzzing procedure to generate and mutate program inputs based on code coverage. This fusion of SFI testing with fuzzing testing is known as SFI-based fuzzing.[17–21] Different SFI-based fuzzing approaches have been developed. Among them, POTUS[17] and FIZZER[18] focus on testing kernel-level drivers but overlook the execution contexts of injected faults and lack input mutation capabilities. iFIZZ[20] targets internet of things (IoT) firmware applications, taking into account the execution contexts of injected faults, but lacking input mutation. FIFUZZ[19], designed for testing user-level applications, considers the execution contexts of injected faults and supporting input mutation. As the contemporary SFI-based fuzzing approach, EH-FUZZ[21] can test both user-level applications and kernel-level modules, and it proposes using error coverage to guide the generation of error sequences.

2.1.2 Concolic execution and hybrid fuzzing. Concolic execution[38,51–55] is a software verification technique that combines concrete execution with symbolic execution. In this approach, concrete inputs to the program are initially marked as symbolic variables. Then, a concolic executor runs the target program according to a specific program input, collects constraints encountered during the execution path, and subsequently creates new program inputs by negating these constraints. The newly generated inputs are typically fed back into the system to explore various execution paths.

While traditional fuzzing is effective at rapidly generating inputs, it generally produces only inputs that lead to execution paths with relatively loose branch conditions.[56] In contrast, concolic execution excels at discovering inputs that lead to execution paths with complex branch conditions.[38] To leverage the strengths of both traditional fuzzing and concolic execution, a hybrid approach—known as hybrid fuzzing[38,52,56–58]—has been developed. In hybrid fuzzing, the concolic executor takes inputs from the fuzzer, generates new program inputs, and feeds them back to the fuzzer. This process enables the exploration of new execution paths, assisting the fuzzer in uncovering paths governed by intricate branch conditions.

2.2 Key insights

By analyzing the locations of error points, we observe a notable correlation among the paths to error points. In particular, many error points share common paths from the program's entry point to the occurrence of the error. This suggests that when the fuzzer separately addresses these error points, it may redundantly explore or solve the common execution paths, potentially diminishing the efficiency of the fuzzer. However, this issue is largely overlooked in existing SFI-based fuzzing methods. For example, in Section 1, we discussed the vulnerability CVE-2013-4332 in the GNU C Library, which leads to integer overflows in the functions memalign, valloc, and pvalloc. The vulnerability in each function arises from manipulating the variable bytes in a way that causes it to exceed the maximum representable value for the integer data type. Hence, we consider each manipulation of the variable bytes in these functions memalign, valloc, and pvalloc as an exploitable error point. Next, let us consider the path relationship of the functions memalign, valloc, and pvalloc. As shown in Code 2, these three functions are within the same switch case structure. This switch case structure is invoked by the function allocate_1, and based on the value of the variable allocation_function, it selects one of the functions memalign, valloc, or pvalloc to execute. Therefore, within this switch case structure, there are three error points that need to be tested, occurring at line 8, line 13, and line 18. Additionally, the calling path for these three error points is common, traversing through … → allocatethread → allocate → allocate_1 → switch case. If a fuzzer is used to individually explore these three error points, it would require generating test cases repeatedly to explore each path. In this paper, we aim to minimize the redundancy in exploring these paths. We strategically cluster these error points and leverage concolic execution to solve the constraints of paths that enable approaching or reaching the cluster. Subsequently, the fuzzer only needs to vary values in the program input minimally to reach different error points. For instance, in Code 2, altering the value of the variable allocation_function would be sufficient.

```
1  // ... -> allocate_thread -> allocate -> allocate_1 -> switch case
2  allocate_1 (void)
3  {
4      switch (allocation_function)
5      {
6          case with_memalign:
7        {
8          void *p = memalign (alignment, allocation_size); // error point 1
9          return (struct allocate_result) {p, alignment};
10       }
11         case with_valloc:
12       {
13         void *p = valloc (allocation_size); // error point 2
14         return (struct allocate_result) {p, page_size};
15       }
16         case with_pvalloc:
17       {
18         void *p = pvalloc (allocation_size); // error point 3
19         return (struct allocate_result) {p, page_size};
20       }
21     }
22 }
```

Code 2 `memalign`, `valloc` and `pvalloc` reside within the same switch case structure.

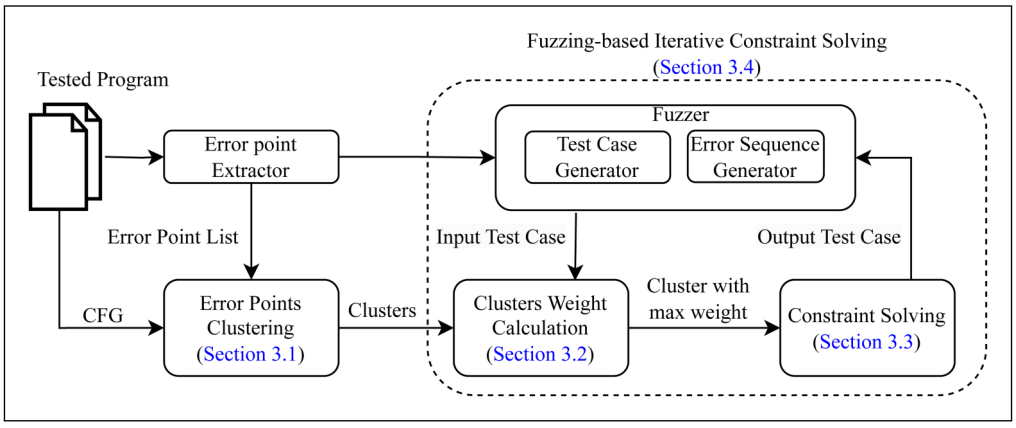

Figure 1. Overall architecture of HuntFUZZ.

Through this clustering strategy, several benefits are achieved:

- **Improved effectiveness in testing error points for SFI-based fuzzing methods.** By reducing the redundant exploration of common paths among error points, HuntFUZZ can test more error sequences within the same timeframe compared to existing SFI-based fuzzing methods. This conclusion is validated in Section 5.2.2. Furthermore, to assess the impact of clustering, we compare the number of error sequences tested with versus without clustering in Section 5.3.1, demonstrating that the clustering method indeed helps in effectively testing more error sequences.
- **Enhanced detection of deep-state error points.** Existing SFI-based fuzzing methods typically rely on traditional fuzzing methods to generate program inputs, which may struggle to test some deep-state error points that depend on highly intricate and specific constraints[38], as discussed in Section 2.1.2. However, HuntFUZZ utilizes concolic execution to strategically solve input constraints within a cluster, which may include deep-state error points. This systematic approach enables the fuzzer to cover such deep-state error points more comprehensively. This is validated in Section 5.2.1.

3 Design of HuntFUZZ

In this section, we explain our design of HuntFUZZ. The overall architecture is illustrated in Figure 1. Firstly, HuntFUZZ statically analyzes the tested program to extract error points using the error point extractor. Like existing SFI-based fuzzers[21], the fuzzer's test case generator then produces program inputs for executing the target program, following a traditional fuzzing approach. The fuzzer gathers the execution status of error points, and the error sequence generator then produces error sequences that indicate if the error points should execute normally (represented as 0) or fail (represented as 1) due to an injected fault, depending on the execution status of the error points. Meanwhile, the fuzzer also gathers runtime information and detects bugs.

In addition to the general flow described above, this paper innovatively introduces the following extra modules:

- **Error points clustering:** This process involves clustering all error points based on the CFG of the tested program and the error point list (generated by the error point extractor). Error points grouped into the same cluster typically share a common path. Moreover, error points within the same cluster tend to have distances from their nearest common parent node that fall within a specific range. We will elaborate on this aspect in Section 3.1.
- **Cluster weight calculation:** In this process, HuntFUZZ chooses a test case generated by the fuzzer and calculates the weight of each cluster based on the number of injected error points in each cluster and the distance from the current path which is determined by this test case. The cluster with the highest weight is then selected as the cluster that the fuzzer aims to reach. We will elaborate on this process in Section 3.2.
- **Constraint solving:** To avoid redundant exploration of error points within a cluster, we provide the fuzzer with optimized test cases. Specifically, based on the cluster with the highest weight obtained from the previous step, the concolic executor then outputs the test cases designed to approach or reach the common parent node of a cluster with the highest weight. We will provide a detailed explanation of this process in Section 3.3.
- **Fuzzing-based iterative constraint solving:** After the concolic executor provides test cases capable of approaching or reaching the current cluster with the highest weight, this iteration continues until either the fuzzer generates

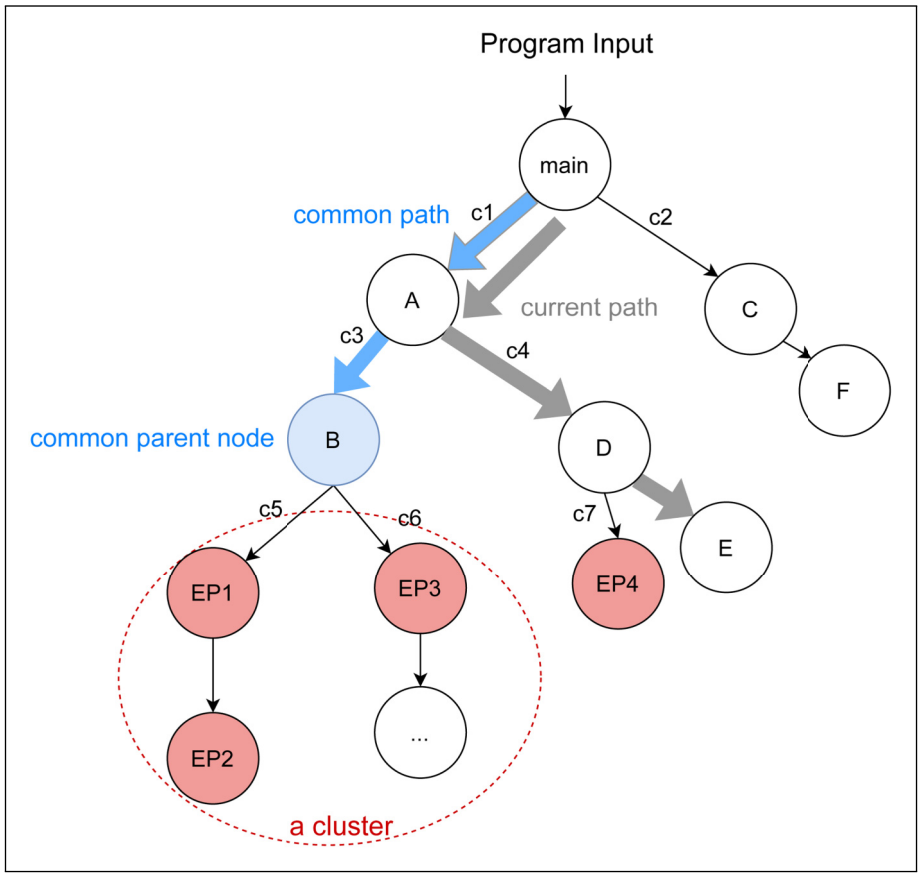

Figure 2. A control flow graph (CFG) of a tested program along with error points that need testing.

test cases that cover all injected error points in the cluster or the number of generated test cases reaches a predefined threshold (indicating that reaching some error points might be difficult). During this iteration, the fuzzer continuously generates inputs. Once the iteration is complete, the concolic executor outputs test cases that can approach or reach the next cluster. We will discuss this process in detail in Section 3.4.

Technical challenges. We pinpoint three challenges in implementing our approach: 1) How to design a clustering method for error points? 2) How to calculate the weight for each cluster? 3) How to design an optimization algorithm to efficiently obtain test cases that approach or reach the cluster?

To illustrate our idea, we provide an example as shown in Figure 2. We schematically illustrate a partial CFG of a tested program along with error points that need testing. Nodes in the CFG represent the basic blocks of the program, so nodes EP1 to EP4 represent the basic blocks where the four tested error points reside. Edges in the CFG represent condition constraints to reach that node. The program's entry point is the main function, which takes program input and executes specific program paths. Suppose a certain program input leads to the program following the path: main → A → D → E, and we want the fuzzer to test error points along other paths. For instance, intuitively, error points EP1, EP2, and EP3, which are closely located and share a common path: main → A → B. To effectively reach these three error points, we want the concolic executor to output test cases approaching or reaching the common parent node B, allowing the fuzzer to mutate these output test cases to generate new test cases. Given their closeness in distance to node B, we expect the fuzzer can easily reach the three error points. In addition to guiding the fuzzer to reach these three error points, this approach also offers the advantage of efficiency, as it reduces redundancy in exploring the common path repetitively for each error point.

3.1 Error points clustering

To identify error points with common paths, in this module, we propose a clustering algorithm for error points. For a given k value, this algorithm ensures that the error points grouped together have a distance from their nearest common parent node that is less than or equal to k. Therefore, k represents the maximum distance between all error points in a cluster

and their (nearest) common parent node. Theoretically, the value of k can be defined as a fixed integer within the range (0, maxDepthEP) (line 4 in Algorithm 1), where maxDepthEP denotes the maximum depth of the error points. However, in our experiments on several real-world programs (discussed in Section 5.3.1), we evaluated cases with different values of k, starting at 1 and incremental increasing by 1. The experimental results indicate error coverage improves from 1 to 2, and then decreases as k increases further (e.g. $k = 3,\ 4,\ 5$ as we tested). Based on these observations, we have selected k=2 as the default value for testing real-world programs in Section 5.

Algorithm 1. Error points clustering

```
Input: error point list E, control flow graph G
Output: error point cluster EPC
procedure getErrorPointCluster (E,G)
 1: EPC ← [ ]
 2: S ← [ ] // indicating whether an error point is visited
 3: bbkSet ← [ ]
 4: k ← fixed_k_value // define fixed_k_value in the range (0, maxDepthEP)
 5: for i ← 0 to length(E) do
 6:    bbk ← getFatherList(E[i], G, k)
 7:    bbkSet[i] ← bbk
 8:    S[i] ← false
 9: end for
10: while existUnvisited(S) do
11:    CEI ← getRandomEP(E, S)
12:    S[CEI] ← true
13:    P ← [ ]
14:    for i ← 0 to length(bbkSet) do
15:       if isSamePATH(S[CEI], S[i]) then
16:          P.add(E[i])
17:          S[i] ← true
18:       else
19:          if hasCommon(bbkSet[CEI], bbkSet[i]) then
20:             P.add(E[i])
21:             S[i] ← true
22:          end if
23:       end if
24:    end for
25:    EPC.add(P)
26: end while
27: return EPC
end procedure
```

As illustrated in Algorithm 1 (for algorithms presented in this paper, the for statements follow the convention of using a half-open interval, similar to Python[59], rather than a closed interval), firstly (lines 5–9), for each error point, we traverse upward for k iterations to get a set of parent nodes. This set is referred to as bbkSet. This functionality is primarily implemented by the function getFatherList, which takes three parameters: E[i], G, and k. E[i] represents an error point; G is the program's CFG where each node corresponds to a basic block; and k is the traversal depth. Thus, the function getFatherList traverses upwards from the basic block containing E[i] in the CFG for up to k levels, recording all traversed basic blocks in bbk. Then, in lines 10–26, we compare the bbkSets of error points. If there are n (n≥ 2) error points sharing a common node within their respective bbkSets, it signifies that the common ancestor's distance from these n error points is less than or equal to k. Consequently, these n error points are clustered together. If this condition is not met, clustering cannot be performed. For instance, in Figure 2, for the error points EP1, EP2, EP3 and EP4, when k = 2, the bbkSets for these four error points are, respectively, B, A, EP1, B, B, A, and D, A. It can be observed that EP1, EP2, and EP3 share a common parent node B, and the distance of these three error points to B is less than or equal to 2. Therefore, we can cluster EP1, EP2, and EP3 into one group. It is worth noting that EP1, EP3, and EP4 also have a common parent node A. However, our algorithm chooses to prioritize clustering EP1 and EP2, which belong to the same path (as shown in lines 15–17 of Algorithm 1), because such error points often share longer common paths. For instance, the common path for EP1, EP2, EP3 is main → A → B, while EP1, EP3, EP4 shares the common path main → A. Clearly, the first clustering method results in error points with longer common paths. Besides, reducing the redundant exploration of longer common

paths implies a greater improvement in the efficiency of the fuzzer. As a result, the final clustering result for these four error points is set1: EP1, EP2, EP3, set2: EP4.

3.2 Cluster weight calculation

Initially, the concolic executor follows the path based on a test case generated by the fuzzer, but this path may not approach or reach the targeted error points. And we need the concolic executor to guide the fuzzer in covering error points within a specific cluster. To determine which cluster to cover, we propose a strategy that prioritizes covering clusters with a higher number of injected error points and clusters that are closer to the current path. The reasons for considering these two characteristics of the cluster are based on the following insights:

- Clusters with a higher number of injected error points represent areas of the program with more frequent or severe issues. By prioritizing these clusters, we focus on the regions of the code most likely to contain critical bugs.
- Prioritizing clusters that are closer to the current path is related to the requirements of the concolic executor. The concolic executor needs to solve constraints to reach a specific cluster. This solving process involves collecting constraints from the current path and deriving the path constraints to the cluster through methods such as negating the constraint (detailed in Algorithm 3 and Section 3.3). Choosing clusters that are closer to the current path facilitates this process, as the concolic executor can more easily collect the constraints needed to reach a nearby cluster compared to a distant one.

Thus, as shown in Algorithm 2, it first retrieves the complete execution path corresponding to a test case (FI) using the getCompletePath function (line 2). Then, for each cluster, it determines the number of injected error points within the cluster and identifies the common parent node (lines 4 and 5). Then, for each node in the complete execution path, it computes the distance from that node to the common parent node of the cluster and accumulates these distances to determine the cluster distance (lines 7–9). In lines 10–16, the algorithm performs a weighted sum of the number of injected error points in the cluster and the inverse of the distance of the cluster, ultimately identifying the cluster with the highest weight (cluster_max).

Algorithm 2. Get cluster with max weight

Input: error point clusters *EPC*, error sequence *ES*, test case generated by fuzzer *FI*
Output: cluster with maximum weight *clusterWeight*
procedure getMaxCluster (*EPC*, *ES*, *FI*)
1: $maxCw \leftarrow 0$
2: $completePath \leftarrow$ getCompletePath(*FI*)
3: **for** $i \leftarrow 0$ **to** length(*EPC*) **do**
4: $EPNum \leftarrow$ getEPNum($EPC[i], ES$)
5: $parentNode \leftarrow$ findCommonParentNode($EPC[i]$)
6: $clusterDistance \leftarrow 0$
7: **for** node in *completePath* **do**
8: $clusterDistance \leftarrow clusterDistance +$ distance(node to *parentNode*)
9: **end for**
10: $clusterWeight \leftarrow w1 \times EPNum + w2 \times clusterDistance^{-1}$
11: **if** $clusterWeight > maxCw$ **then**
12: $maxCw \leftarrow clusterWeight$
13: $cluster_{max} \leftarrow EPC[i]$
14: **end if**
15: **end for**
16: **return** $cluster_{max}$
end procedure

3.3 Constraint solving

In this section, we discuss how the concolic executor generates test cases capable of approaching or reaching a cluster_max (or its common parent node). Specifically, while executing the current path based on the test case generated by the fuzzer, the concolic executor collects constraints for this path. However, the current path might not reach the target common parent node. Then, the concolic executor negates the constraints of the specific conditional branch executed by the current

Algorithm 3. Constraint solving

```
Input: test case generated by fuzzer FI, cluster_max
Output: testcases generated by concolic executor
procedure constraintSolving (FI, cluster_max)
 1: parentNode ← findCommonParentNode(cluster_max)
 2: currentPath ← getCurrentPath(FI)
 3: testcases ← []
 4: while FI.unDone() do
 5:   if currentPath.reached(parentNode) then
 6:     break
 7:   end if
 8:   curInst = currentPath.curInst()
 9:   if curInst.isConditionalJmp() then
10:     originDistanceToCluster_max ← getDistanceToClusterMax(curInst.getNext(),parentNode)
11:     negateDistanceToCluster_max ← getDistanceToClusterMax(curInst.getNegate(),parentNode)
12:     if originDistanceToCluster_max > negateDistanceToCluster_max then
13:       curInstConstraint = curInst.getConstraint()
14:       negatedPathConstraint = currentPath.getConstraint().add(negate(curInstConstraint))
15:       if isSatisfiable(negatedPathConstraint) then
16:         testcases.add(exploreNewPath(negatedPathConstraint))
17:       else
18:         if isSatisfiable(negatedPathConstraint) then
19:           testcases.add(exploreNewPath(negatedPathConstraint))
20:         end if
21:       end if
22:     end if
23:   end if
24:   FI.goOn()
25: end while
26: return testcases
end procedure
```

instruction and explores new paths to approach or reach the target common parent node. It is worth noting that exploring new paths by negating constraints and constraint solving is a common strategy in concolic execution.[38] Our innovation lies in applying this strategy to explore clusters of error points.

In detail, as shown in Algorithm 3, this process begins by identifying the common parent node (parentNode) of the target cluster cluster_max, getting the current execution path (currentPath) of FI, and an empty list is initialized to store test cases generated by the concolic executor (lines 1–3). The algorithm processes the execution of the fuzzer-generated test case FI, examining each instruction. If the current execution path reaches the common parent node (parentNode), the iteration concludes (lines 5–7). For conditional jump instructions, the algorithm uses curInst.getNext() to retrieve the original next instruction (the non-negated path) and compute its distance to the target parentNode, storing the result as originDistanceToCluster_max (line 10). Here, the distance is calculated as the distance between the basic block containing the next instruction (curInst.getNext()) and the basic block containing parentNode. Similarly, the algorithm uses curInst.getNegate() to retrieve the instruction executed after the negated conditional jump instruction (the negated path) and computes its distance to the target parentNode, storing the result as negateDistanceToCluster_max (line 11). If originDistanceToCluster_max is greater than negateDistanceToCluster_max, this indicates that the negated path is closer to the parentNode, and the condition expression should be negated. Thus, the algorithm adds a negated instruction constraint (negate(curInstConstraint)) to the current path constraint (currentPath.getConstraint()) and attempts to solve the updated constraint (`negatedPath-Constraint`) using a constraint solver (line 14). It is worth noting that when constructing the negated constraint of conditional jump instructions, the algorithm negates the entire expression of the conditional jump instruction. For example, if the expression of a conditional jump instruction is $p_1 \wedge p_2$, the negated instruction constraint is constructed as $\neg(p_1 \wedge p_2)$.

If the negatedPathConstraint is satisfiable, it means test cases can be generated to explore new paths. If the solver fails to find a solution on the first attempt, the algorithm attempts to solve it again (lines 15–21). The major reason for the failure in the first attempt is the timeout, which has also been reported in the related work.[60] If it still fails, the algorithm proceeds with executing FI (line 24) until another conditional jump instruction is encountered and meets the criteria for negation, at which point it will attempt to solve the constraints again. If all the negatedPathConstraint during the execution of FI cannot be solved, the algorithm will return an empty test case.

3.4 Fuzzing-based iterative constraint solving

Algorithm 4. Fuzzing-based iterative constraint solving

Input: tested program P, all error-points location info EP, error sequence ES, test case generated by fuzzer FI
Output: *testcases*

```
1:  G ← getCFG(P)
2:  E ← getErrorPointList(EP)
3:  EPC ← getErrorPointCluster(E, G)
4:  TP ← getInjectedErrorPointPath(ES, G, EP)
5:  curMutationCount ← 0
6:  cluster_max ← getMaxCluster(EPC, ES, FI)
7:  coverErrorFlag ← []
8:  concolic executor generates testcases, testcases ← constraintSolving(FI, cluster_max)
9:  fuzzer reads testcases
10: while (fuzzer updates FI) do
11:   if curMutationCount > mutateThreshold or clusterState == covered then
12:     remove currentCluster from EPC
13:     concolic executor reads FI
14:     nextCluster ← getMaxCluster(EPC, ES, FI)
15:     testcases ← constraintSolving(FI, nextCluster)
16:     fuzzer reads testcases
17:     curMutationCount ← 0
18:     continue
19:   end if
20:   curMutationCount ← curMutationCount + 1
21:   for i ← 0 to length(TP) do
22:     if (executing FI covers TP[i]) then
23:       coverErrorFlag[i] ← true
24:     end if
25:   end for
26: end while
```

We define the criteria for determining the completion of a cluster detection iteration as whether the fuzzer is covering all the injected error points within the cluster or if the number of test cases generated by the fuzzer exceeds a certain threshold. In the latter case, we may consider that some error points within the cluster are too challenging to be covered. Once the concolic executor generates test cases capable of approaching or reaching a cluster_max (as shown in Algorithm 3), subsequently, the fuzzer continuously generates inputs until the completion of this cluster detection.

Therefore, the process of an iteration of the constraint solving and fuzzer is depicted in Algorithm 4. For lines 1–9, it initializes a CFG (G), error point list (E), error point cluster (EPC, based on Algorithm 1), and injected error point path (TP). Simultaneously, it initializes the count of inputs generated by the fuzzer (curMutationCount), the cluster of maximum weight (cluster_max, based on Algorithm 2). Then, the concolic executor generates test cases by solving constraints (based on Algorithm 3), and the generated test cases are stored as files in a designated directory. The fuzzer reads the test cases generated by the concolic executor from this directory and performs mutations on these test cases to generate new ones. Subsequently, lines 10–26 constitute the main loop of fuzzing-based constraint solving. In this process, the fuzzer continuously generates inputs. Within this flow, lines 11–19 indicate that if all injected error points in a cluster are covered or if the number of fuzzer-generated inputs reaches mutateThreshold (indicating that generating inputs to cover some error points is deemed challenging), the concolic executor selects the last tested test case (i.e. updated FI) from the test case queue, where the FI determines the current execution path, and based on this path, calculates the next cluster with the highest weight and then generates test cases that approach or reach the cluster. Lines 21–25 signify the verification of whether the fuzzer-generated input covers the error point. If coverage is achieved, the error point coverage status is updated. If not, the fuzzer proceeds to generate test cases to cover the uncovered error points within the cluster.

4 Implementation

In this section, we elaborate on the details of implementing HuntFUZZ, covering three main aspects: error points extractor, static code instrumentation, runtime fuzzing, and concolic executor.

4.1 Error points extractor

Our approach extracts function calls as error points, as recent studies[19,21,61] indicate that the majority of error points involve code statements checking error-indicating return values of function calls. We identify candidate error points by examining functions that return pointers or integers, following the method used in EH-FUZZ.[21] Additionally, we employ nine distinct exception-handling methods to aid in the recognition of error-handling functions. These methods include four categories implemented through jump branching statements, including return, break, continue, and goto. The other five categories involve functions that handle exceptional states, such as logging (log), program termination (exit), closing files or directories (close), deleting files or directories (delete), and freeing memory (free).

4.2 Static code instrumentation

Before initiating the runtime fuzzing and concolic execution phases, it is essential to record program-related information to the runtime stage. Thus, during the compilation phase, we instrument the following four types of information:

- **Basic blocks information:** To enable the fuzzer to collect code coverage, we instrument the basic blocks information. This instrumentation method assigns a unique ID to each basic block, allowing the program to capture edge coverage information between two basic blocks when it transitions from one to another during execution.
- **Context information:** In HuntFUZZ, the fuzzer needs to record the runtime context of each error point, as different contexts may lead to different error circumstances.[21] This context information is represented as the function call relationships of the error points. To record this information, we insert a monitoring code at the entry and exit points of each function.
- **Error point information:** To effectively handle error points, we instrument detailed information about them, including their precise locations and the mechanisms for fault injection. Fault injection involves skipping the function call at the error point and assigning its return value to abnormal or error-prone values, such as a null pointer or a random negative integer. The execution of the fault injection code is controlled by a conditional (if-branch) statement. Specifically, the program evaluates a flag variable (corresponding to the error sequence) within the conditional. If the flag is set, the original function call is bypassed, and the fault injection code is executed. Otherwise, the normal execution flow is maintained.
- **Cluster information:** For each cluster, we assign a unique ID that acts as a prefix for the error points within the cluster, allowing fuzzer and concolic executor to identify error points and their common parent nodes within the cluster.

To implement the aforementioned instrumentation, we use LLVM[62] to modify the program's intermediate representation files, ultimately generating an instrumented binary.

4.3 Runtime fuzzing and concolic executor

To clearly describe the implementation of the runtime fuzzing and concolic executor, we identify and label nine modules (modules ❶–❾) in Figure 3. First, the instrumented target program (as described in Section 4.2) is provided as input to both the fuzzer and the concolic executor. Next, we will detail how the fuzzer and concolic executor are deployed.

Fuzzer. In the fuzzer, the test case generator (module ❶) utilizes AFL's[34] test case generation method to create inputs, which are then written into the test case queue (module ❸, a directory). Beyond that, the fuzzer reads output test cases from the queue that are produced by the concolic executor and have approached or reached the common parent node of the target cluster. Based on these output test cases, the test case generator mutates them further to create new test cases, aiming to cover injected error points within the cluster. The error sequence generator (module ❷) produces error sequences to determine whether a fault should be injected (same as EH-FUZZ[21]). The locations and context information (an array that records the runtime function call stack for each error point) of these error points are recorded in the FaultInjectRecorder object. The generated error sequences are stored in rawInjectMap, a shared memory variable (module ❹), which is also read by the concolic executor to detect fault injection status. Finally, the fuzzer employs bug checkers (module ❾), such as ASan[63] and MSan[64], to analyze runtime memory information and determine if any bugs have been triggered.

Concolic executor. The concolic executor is implemented based on Pin.[65] Specifically, the concolic executor reads the input test cases and error sequence generated by the fuzzer from the test case queue (module ❸) and shared memory variable (module ❹), respectively. It then selects the untested cluster with the highest weight according to Algorithm 2 (module ❺). Using Pin's dynamic instrumentation Application Programming Interface, the concolic executor performs instruction-level dynamic instrumentation on the current execution path (module ❻). The type of instrumentation applied

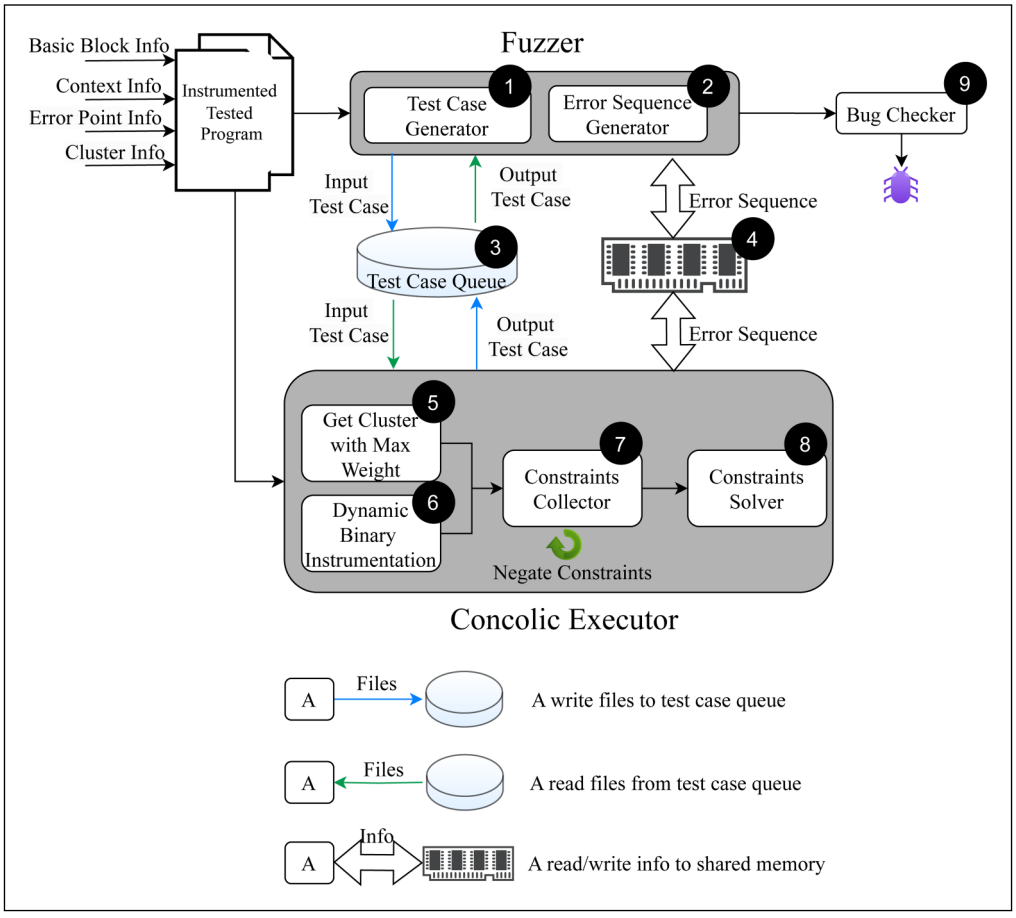

Figure 3. Implementation details of runtime fuzzing and concolic executor.

depends on the type of instruction being executed, which determines the corresponding callback function. Specifically, the analyzeTrace function is used as the callback function for instruction-level instrumentation. This function defines different callback functions for different instruction types. For instance, if the program reaches a conditional jump instruction, the analyzeJmp function will be called to instrument the relevant functionality code for constraint collection and solving. As the current path is executed, the constraints collector (module ❼) gathers and adds the path constraints (implemented in the getExprFromReg function). The concolic executor monitors whether the distance from the current path to the target cluster increases. If such an increase is detected, it negates the constraints and explores alternative paths, as outlined in Algorithm 3. Once the path constraints leading to the common parent node of the target cluster are collected, the z3 solver[66] (module ❽) generates test cases that satisfy these constraints. These test cases are then output into the test case queue.

5 Evaluation

In this section, we evaluate the following questions:

- How effective is HuntFUZZ in discovering bugs in real-world applications? Can it discover zero-day bugs?
- How does HuntFUZZ perform compared to other state-of-the-art fuzzing approaches in terms of bug finding, error coverage, and code coverage?
- How do the parameters associated with the algorithm(s) influence the overarching efficacy of the HuntFUZZ framework?
- How effective is HuntFUZZ in covering clusters, and what percentage of error points within the clusters are covered? Based on the test cases output by the concolic executor, what percentage of the test cases generated by the fuzzer can reach the target cluster_max, and what percentage of the test cases are unable to reach any cluster?

Experimental environment and setup. We conducted our experiments on a machine powered by an Intel(R) Xeon(R) Gold 5118 CPU2.30 GHz with 16 cores. The experiments are performed on an Ubuntu 20.04.5 LTS operating system. To validate HuntFUZZ, we evaluate it on two datasets (UniBench[41] and applications previously tested by EH-FUZZ[21]), as well as nine non-benchmark applications. UniBench comprises 20 test applications, while the EH-FUZZ benchmark consists of 15 test applications. Due to the duplication of two applications across these two datasets, we conducted a total of 42 application tests. The basic information of these applications is listed in Table 1.

Error point identification. For the tested applications, we first utilize HuntFUZZ to statically analyze their source code, identifying potential error points. Subsequently, we manually identify realistic error points capable of causing failures and errors. Table 1 displays the error points recognized by HuntFUZZ. Overall, HuntFUZZ identifies 18,213 error points. Among them, we manually confirm these error points and ultimately determine 10,684 realistic error points. Indeed, the manual selection of realistic error points is not challenging, as many error points call the same functions.

5.1 Found bugs

For the error points indicated in Table 1, we utilize HuntFUZZ to conduct testing on them. HuntFUZZ tests each program using ASan[63] and MSan[64] to detect bugs, limiting the testing time to 24 h, and repeating the experiment five times. The results of the bugs we found are shown in Table 1. Overall, HuntFUZZ has discovered 162 known bugs, among which 62 are related to error handling. For bugs leading to program crashes or failures, we manually examine their root causes using bug reports and source code to determine whether they are known bugs or unique zero-day bugs. Notably, HuntFUZZ discovered seven zero-day bugs in Jasper, libtiff, OpenSSL, tidy, jqlang, bash, and mksh. We have responsibly reported these zero-day bugs. The zero-day bug in libtiff has been confirmed by the developers. The zero-day bug in jqlang is also simultaneously found by OSS-FUZZ[76] and this bug has already been fixed. We are awaiting responses regarding the other bugs.

Here, we provide a detailed overview of the zero-day bugs discovered in Jasper, libtiff, and OpenSSL. The detailed information about the other zero-day bugs is shown in Appendix A.

Wild free bug in Jasper. In Code 3, the function jas_iccprof_create makes use of jas_malloc (line 5) and jas_iccattrtab_create (line 9) to check whether the variables prof and prof->attrtab are allocated correctly, with both jas_malloc and `jas_iccattrtab _create` encapsulating the malloc function. At this point, upon detecting that prof is successfully allocated (prof $\neq$ NULL) while prof->attrtab allocation fails (prof->attrtab = NULL), the program proceeds to line 14, entering error handling. Consequently, lines 11–12 are not executed, leaving the variable prof->tagtab.ents uninitialized. However, the error handling code (lines 14–17) invokes jas_iccprof_destroy, and due to the uninitialized prof->tagtab.ents, when attempting to free prof->tagtab.ents (line 25), a wild free bug occurs.

NULL-pointer dereference bug in libtiff. In Code 4, within the function TIFFReadDirectory, there is an if statement that checks whether the return value of the function _TIFFMergeFieldInfo is NULL (line 4). The second parameter of _TIFFMergeFieldInfo is the return value of _TIFFCreateAnonFieldInfo. The function _TIFFCreateAnonFieldInfo uses `_TIFFmal-loc` to allocate memory for the variables fld and fld->field_name (lines 24 and 27). When there is a failure in allocating memory for either of these variables, the return value of _TIFFCreateAnonFieldInfo is NULL. In such a scenario, calling _TIFFMergeFieldInfo (line 4) leads to a NULL pointer dereference bug.

```
1  // jas_iccprof_create -> jas_iccprof_destroy -> jas_free
2  static jas_iccprof_t *jas_iccprof_create()
3  {
4      if (!(prof = jas_malloc(sizeof(jas_iccprof_t))))
5      {
6          goto error;
7      }
8      if (!(prof->attrtab = jas_iccattrtab_create()))
9          goto error;
10     prof->tagtab.numents = 0;
11     prof->tagtab.ents = 0;
12     return prof;
13 error:
14     if (prof)
15         jas_iccprof_destroy(prof);
16     return 0;
17 }
18 void jas_iccprof_destroy(jas_iccprof_t *prof)
19 {
20     if (prof->attrtab)
```

Table 1. Information about the applications tested with HuntFUZZ, as well as the identified error points and bug information found by HuntFUZZ.

	Tested program	Version	Identified error points	Realistic error points	Known bugs	Error handling bugs	Zero-day bugs
Unibench[41]	exiv2	0.26	136	70	3	2	0
	gdk-pixbuf-pixdata	gdk-pixbuf 2.31.1	107	63	2	0	0
	Jasper	2.0.12+2.0.14	227	92	4	2	1
	jhead	3.00	530	359	5	3	0
	libtiff	3.9.7+4.5.1	985	695	7	3	1
	lame	3.99.5	830	332	5	1	0
	mp3gain	1.5.2	273	198	1	0	0
	swftools	0.9.2	674	571	3	0	0
	ffmpeg	4.0.1	198	112	13	7	0
	flvmeta	1.2.1	636	254	2	0	0
	Bento4	1.5.1−628	581	348	6	2	0
	cflow	1.6 + 1.7	117	88	1	0	0
	ncurses	6.1	525	210	3	0	0
	jq	1.5	618	485	6	2	0
	mujs	1.0.2	431	279	2	0	0
	pdftotext	4.00	328	165	3	1	0
	SQLite	3.8.9	153	91	5	3	0
	binutils	2.28	362	144	6	2	0
	libpcap	1.8.1	731	329	4	1	0
	tcpdump	4.8.1	912	626	9	3	0
EH-FUZZ[21]	vim	8.2.3595	334	270	5	2	0
	bison	3.8.1	187	125	0	0	0
	nasm	2.15.05	62	26	0	0	0
	catdoc	0.95	101	69	5	4	0
	clamav	0.104.1	2125	1247	3	1	0
	gif2png+libpng	2.5.14 + 1.6.3	129	65	0	0	0
	OpenSSL	3.0.0 + 3.0.9	135	102	4	3	1
	btrfs	Linux 5.16.16	929	351	3	1	0
	xfs	Linux 5.16.16	201	171	1	1	0
	jfs	Linux 5.16.16	114	100	2	1	0
	cephfs	Linux 5.16.16	460	140	4	3	0
	xhci	Linux 5.16.16	180	104	1	1	0
	vmxnet3	Linux 5.16.16	98	43	3	1	0
Non-benchmark program	man-db[67]	2.12.0	295	158	5	2	0
	woff2[68]	1.0.2	163	139	3	0	0
	gzip[69]	1.13	397	272	6	2	0
	bzip2[70]	1.0.6	432	365	5	2	0
	sassc[71]	3.6.2	321	284	3	0	0
	tidy[72]	5.9.20	527	381	2	0	1
	jqlang[73]	1.7	118	67	3	1	1
	bash[74]	5.2.21	827	351	8	3	1
	mksh[75]	mksh-R59c	724	343	6	2	1
Total	42		18,213	10,684	162	62	7

```
        jas_iccattrtab_destroy(prof->attrtab);
    if (prof->tagtab.ents)
        jas_free(prof->tagtab.ents);
    jas_free(prof);
}
void jas_free(void *ptr)
{
    free(ptr);
}
```

Code 3 Wild free bug in Jasper.

```
// TIFFReadDirectory -> _TIFFMergeFieldInfo -> _TIFFCreateAnonFieldInfo -> ...-> _TIFFmalloc
TIFFReadDirectory(TIFF* tif)
{
    if (!_TIFFMergeFieldInfo(tif, _TIFFCreateAnonFieldInfo(tif, dp->tdir_tag, (TIFFDataType) dp
    ->tdir_type),1))
}
int _TIFFMergeFieldInfo(TIFF* tif, const TIFFFieldInfo info[], int n)
{
    for (i = 0; i < n; i++)
    {
        const TIFFFieldInfo *fip =
            _TIFFFindFieldInfo(tif, info[i].field_tag, info[i].field_type);
        if (!fip) {
            *tp++ = (TIFFFieldInfo*) (info + i);
            tif->tif_nfields++;
        }
    }
    return n;
}
TIFFFieldInfo* _TIFFCreateAnonFieldInfo(...)
{
    fld = (TIFFFieldInfo *) _TIFFmalloc(sizeof (TIFFFieldInfo));
    if (fld == NULL)
        return NULL;
    fld->field_name = (char *) _TIFFmalloc(32);
    if (fld->field_name == NULL)
    {
        return NULL;
    }
}
void* _TIFFmalloc(tsize_t s)
{
    return (malloc((size_t) s));
}
```

Code 4 NULL-pointer dereference bug in `libtiff`.

NULL-pointer dereference bug in OpenSSL. When testing the OpenSSL custom module X509 with insufficiently allocated space for X509, it can result in the function do_cmd calling the function lh_FUNCTION_retrieve, which sets the value of the variable fp to NULL (line 5, Code 5). Subsequently, when invoking the function EVP_get_digestbyname in line 8, it leads to the execution of the function ossl_lib_ctx_get_data (line 1). The function ossl_lib_ctx_get_data is responsible for retrieving context information (line 15) and dereferencing the variable ctx->lock (line 16). Besides, the function context_init initializes the structure variable ctx. When initialization fails, the error handling code is executed, setting all fields of ctx to NULL. This results in a NULL-pointer dereference bug when dereferencing the variable ctx->lock in line 16.

```
// do_cmd -> lh_FUNCTION_retrieve -> EVP_get_digestbyname ->...-> ossl_lib_ctx_get_data
static int do_cmd()
{
    //fp: retrieve function pointer
    fp = lh_FUNCTION_retrieve(prog, &f);
    if (fp == NULL)
    {
        if (EVP_get_digestbyname(argv[0])) {...}
```

```
9     }
10    return 1;
11 }
12 void *ossl_lib_ctx_get_data()
13 {
14    ctx = ossl_lib_ctx_get_concrete(ctx);
15    if (!CRYPTO_THREAD_read_lock(ctx->lock))
16        return NULL;
17 }
18 static int context_init(OSSL_LIB_CTX *ctx)
19 {
20    ctx->oncelock = CRYPTO_THREAD_lock_new();
21    if (ctx->oncelock == NULL)
22        goto err;
23    return 1;
24  err:
25    memset(ctx, '\0', sizeof(*ctx));
26    return 0;
27 }
```

Code 5 NULL-pointer dereference bug in `OpenSSL`.

Bug features. We attribute HuntFUZZ's ability to discover zero-day bugs to its capacity to achieve higher error coverage than other fuzzers within the same timeframe (refer to Section 5.2.2). Error coverage signifies the fuzzer's proficiency in thoroughly testing scenarios involving the injection of errors. Reviewing these zero-day bugs, three key observations emerge: 1) most of the errors caused by these bugs revolve around operations on pointer-type data. For example, Jasper's bug involves a wild free operation on an uninitialized pointer, while the other two involve dereference operations on NULL pointers. This suggests that incorrect operations on pointers are prone to triggering program crashes or failures; 2) we find that Jasper and OpenSSL bugs result from incorrect error handling functionality, while the bug in libtiff is caused by a failed malloc operation. This indicates that our tool can detect not only bugs related to error handling but also other types of bugs leading to program crashes or failures; 3) we find that some zero-day bugs require the simultaneous activation of two error points to trigger. For example, for Jasper, there are two error points: one where prof = jas_malloc(sizeof(jas_iccprof_t)) (line 5 in Code 3), and the other where prof->attrtab ≠ jas_iccattrtab_create() (line 9 in Code 3). The bug in libtiff has two error points: fld = NULL (line 24 in Code 4) and fld->field_name = NULL (line 27 in Code 4). The OpenSSL bug also has two error points: fp = NULL (line 5 in Code 5) and ctx->lock = NULL (line 27 in Code 5).

5.2 Comparison to existing fuzzing approaches

We select four state-of-the-art fuzzing approaches for comparison on testing 33 applications from two datasets (Unibench [41] and applications tested by EH-FUZZ[21]), including three traditional fuzzers (AFL[34], AFL++[42], and AFLGo[43]) and one SFI-based fuzzer: EH-FUZZ.[21] It is worth noting that within the current landscape of SFI-based fuzzing approaches[17–21], both POTUS[17] and iFIZZ[20] are limited to testing specific domains of applications. Specifically, POTUS is tailored for USB driver testing, while iFIZZ is designed for testing IoT firmware applications. Since these tools do not align with the applications we intend to test, and to our knowledge, FIZZER[18] and FIFUZZ[19] are not yet open source, we ultimately opt for EH-FUZZ as the tool for comparison with HuntFUZZ. We compare HuntFUZZ with selected/representative fuzzing tools in terms of bug finding, error coverage (the number of error sequences), and code coverage (the number of covered code branches).

In Table 2, we present the results of the comparative experiments. For each tool, we repeat the experiment five times. The number of bugs detected by the tool is the total number from all five runs, while the values for branches and error sequences represent the average across these runs. The depth column indicates the number of error-handling bugs identified by the tool, where the depths of the error points that trigger these bugs fall within a specific range. For HuntFUZZ, we configure the parameter values associated with the algorithm at their default value, that is, k = 2, w1 = w2 = 0.5, and mutateThreshold = 10,000.

5.2.1 Comparison on bug finding. Due to the fact that AFL[34], AFL++[42], and AFLGo[43] are only capable of testing user-level applications, they are utilized to assess the user-level applications listed in Table 2. Since AFLGo is a directed fuzzer[43], which means it focuses on guiding the fuzzing process toward specific parts of the code rather than exploring the entire code randomly, we use the line of the identified error points as AFLGo's targets. However, no faults are injected at these error points. It is observed that AFL++ and AFLGo outperform AFL in discovering more bugs, owing to their

Table 2. The results of comparing HuntFUZZ with four state-of-the-art fuzzing approaches in terms of bug finding, error coverage, and code coverage.

	AFL		AFL++		AFLGo		EH-FUZZ			HuntFUZZ						
								Depth					Depth			
Tested program	Bug	Branch	Bug	Branch	Bug	Branch	Bug (error handling bug)	< 500	> = 500	Branch	ErrSeq	Bug (error handling bug)	< 500	>=500	Branch	ErrSeq
exiv2	0	5636	0	17,497	0	12,945	1(1)	1	0	26,484	23,741	3(2)	1	1	11,353	35,372
gdk_pixbuf_pixdata	0	9256	0	14,536	0	14,823	0	0	0	17,256	44,841	2(0)	0	0	7821	58,952
jasper	0	7904	0	11,253	0	12,553	2(1)	1	0	15,904	5684	4(2)	1	1	6978	7023
jhead	0	9953	0	9952	0	10,234	2(1)	1	0	15,564	7831	5(3)	1	2	8342	9102
libtiff	0	4246	0	29,547	2	11,568	3(1)	1	0	36,850	30,495	7(3)	2	1	14,109	44,051
lame	0	5242	0	26,234	0	19,835	1(0)	0	0	27,230	961	5(1)	0	1	15,967	1237
mp3gain	0	7254	0	10,632	0	9894	0	0	0	17,546	213	1(0)	0	0	9448	1149
swftools	0	6351	0	16,569	0	13,056	0	0	0	21,654	24,510	3(0)	0	0	12,305	36,294
ffmpeg	0	5127	2	13,246	3	12,065	10(5)	4	1	25,312	12,368	13(7)	4	3	10,320	19,376
flvmeta	0	4465	0	14,254	0	10,934	1(0)	0	0	16,446	2984	2(0)	0	0	8295	4208
Bento4	0	9962	0	22,156	0	22,105	2(1)	1	0	24,304	5692	6(2)	1	1	17,730	8843
cflow	0	2258	0	15,061	0	15,964	1(1)	1	0	16,218	367	1(0)	0	0	4692	992
ncurses	0	3433	0	23,972	0	19,127	1(1)	1	0	28,338	2072	3(0)	0	0	19,934	3015
jq	0	4549	0	34,501	0	12,257	4(2)	2	0	35,554	3591	6(2)	2	0	12,413	4218
mujs	0	2542	0	8839	0	10,213	1(0)	0	0	12,240	1536	2(0)	0	0	6491	2059
pdftotext	0	3807	0	19,956	0	14,247	1(1)	1	0	28,723	2985	3(1)	1	0	14,302	3645
SQLite	0	5154	0	27,562	0	23,395	4(2)	2	0	31,154	3051	5(3)	2	1	19,934	4959
binutils	0	4450	0	16,749	0	19,452	3(2)	2	0	24,601	3774	6(2)	2	0	10,556	4095
libpcap	0	3924	0	12,985	0	19,576	3(1)	1	0	13,024	3352	4(1)	1	0	9560	3857
tcpdump	0	3123	0	20,121	2	18,755	7(3)	3	0	24,702	1976	9(3)	3	0	11,345	2975
vim	0	5937	0	19,305	0	12,958	3(1)	1	0	28,317	23,968	5(2)	1	1	10,851	35,524
bison	0	4085	0	14,521	0	11,975	0	0	0	17,022	11,956	0	0	0	10,835	43,877
nasm	0	4366	0	7834	0	8347	0	0	0	10,118	1285	0	0	0	6431	3102
catdoc	1	586	2	675	0	663	2(1)	1	0	1998	821	5(4)	1	3	759	1070
clamav	0	6961	0	17,140	0	13,125	3(1)	1	0	19,903	13,124	3(1)	1	0	12,145	16,832
gif2png+libpng	0	5167	0	4246	0	3452	0	0	0	7123	53	0	0	0	3245	192
openssl	0	7835	0	12,249	1	13,125	3(1)	1	0	26,484	24,536	4(3)	1	2	8240	37,684
btrfs	–	–	–	–	–	–	2(1)	1	0	11,235	894	3(1)	1	0	1207	1052
xfs	–	–	–	–	–	–	0	0	0	23,845	1042	1(1)	0	1	3481	2154
jfs	–	–	–	–	–	–	1(1)	1	0	8459	1230	2(3)	1	2	2895	2665
cephfs	–	–	–	–	–	–	3(2)	2	0	12,395	739	4(1)	0	1	1968	1549
xhci	–	–	–	–	–	–	0	0	0	4292	1293	1(1)	0	1	3863	2705
vmxnet3	–	–	–	–	–	–	2(1)	1	0	2153	1346	3(1)	1	0	4017	3723
Total	1	143,573	4	441,592	8	366,643	66(32)	31(96.9%)	1(3.1%)	632,448	264,311	121(50)	28(56%)	22(44%)	301,832	407,551

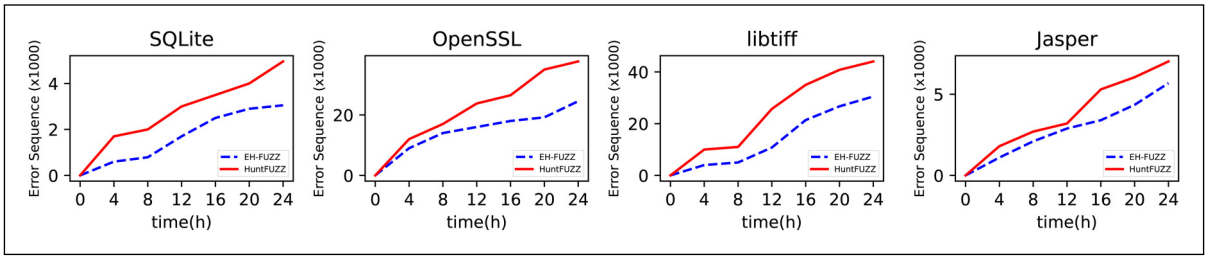

Figure 4. Comparsion of EH-FUZZ and HuntFUZZ in terms of error coverage.

integration of superior strategies for input mutation and seed selection. However, due to the absence of injection error points in these three fuzzing methods, they face challenges in detecting bugs related to error handling. Throughout our testing process, these three tools do not identify bugs associated with error handling.

Compared to the three aforementioned fuzzers, EH-FUZZ[21] and HuntFUZZ both have the capability to test kernel-level applications. Overall, for the user-level and kernel-level applications listed in Table 2, HuntFUZZ has demonstrated the discovery of a greater number of bugs compared to EH-FUZZ, particularly in the realm of error-handling bugs. Moreover, HuntFUZZ identifies all the error-handling bugs detected by EH-FUZZ.

In addition, regarding EH-FUZZ and HuntFUZZ, we summarize the depths of error points that trigger error handling bugs in the CFG. We find that out of the 32 error-handling bugs discovered by EH-FUZZ, depths of 31 bugs' error points <500. Conversely, among the 50 error handling bugs found by HuntFUZZ, depths of 22 bugs' error points ≥500. This finding demonstrates that HuntFUZZ has the ability to test error points with deeper depth. We believe this is because, for some deep-state error points, the program inputs must adhere to very intricate and specific constraints. EH-FUZZ, using traditional fuzzing methods to generate inputs, may struggle to test these deep-state error points (as we discussed in Section 2.1.2). In contrast, HuntFUZZ leverages concolic execution to purposefully solve input constraints within a cluster, which can include deep-state error points. This helps the fuzzer systematically cover such deep-state error points.

5.2.2 Comparison on error coverage. Since the three traditional fuzzing approaches (AFL, AFL++, and AFLGo) cannot conduct fault injection, we compare HuntFUZZ with the representative SFI-based fuzzing method (EH-FUZZ) in terms of error coverage. The results are shown in Table 2. Similar to EH-FUZZ[21], in this paper, error coverage represents the number of error sequences (indicating whether the error points can execute normally or fail). The ability to test more error sequences signifies that the fuzzer can test more scenarios where errors are injected. As shown in Figure 4, we select four applications—SQLite, OpenSSL, libtiff, and Jasper—to showcase the number of error sequences tested by both EH-FUZZ and HuntFUZZ. We can see that HuntFUZZ achieves higher error coverage because the clustering enables fuzzing at a faster pace. Figure 4 shows that HuntFUZZ rapidly ramps up to reach error points, surpassing EH-FUZZ. For example, when testing SQLite, HuntFUZZ can test approximately 2,000 error sequences within about 8 h. In contrast, EH-FUZZ takes around 16 h to test the same number of error sequences (HuntFUZZ has an efficiency improvement of roughly double). Notably, averaging across experiments spanning 24 h for each program test, HuntFuzz achieves 38.9% higher than EH-FUZZ.

5.2.3 Comparison on code coverage. For code coverage, we compare this metric by summarizing the number of code branches tested by various fuzzers. As shown in Table 2, we record the number of code branches tested by the five fuzzers. To further illustrate this, Figure 5 presents how the code branches evolve over time for four selected applications—SQLite, OpenSSL, libtiff, and Jasper—under the influence of these five fuzzing approaches. Generally, EH-FUZZ achieves higher code coverage compared to the three traditional fuzzing approaches. This is because EH-FUZZ covers error points in different calling contexts, which encourages it to explore more code branches. However, this also leads EH-FUZZ to explore some branches that are unrelated to error points. In contrast, HuntFUZZ does not achieve the highest code coverage. This is because HuntFUZZ focuses specifically on code branches where error points reside, without exploring branches that do not contain error points.

5.3 The impact of parameters in algorithms

In this section, we explore the impact of several parameters of the algorithm on the error coverage of HuntFUZZ. We conduct these experiments on seven applications, including SQLite, OpenSSL, libtiff, Jasper, jhead, ffmpeg, and libpcap. These parameters include the distance parameter k in the error points clustering algorithm (Algorithm 1), the weighted

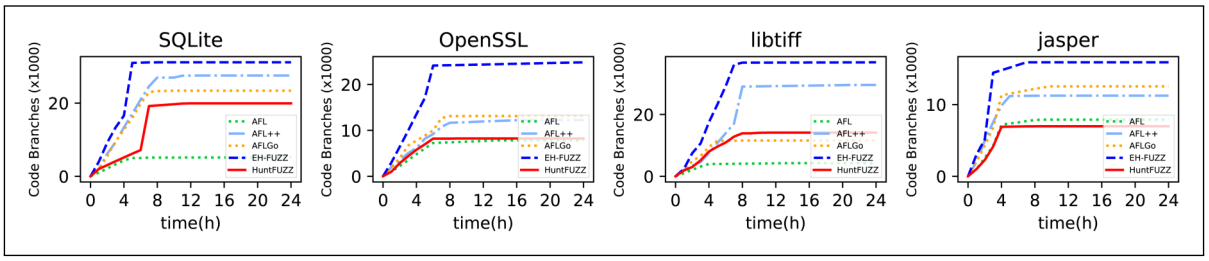

Figure 5. Comparsion of AFL, AFL++, AFLGo, EH-FUZZ, and HuntFUZZ in terms of code coverage.

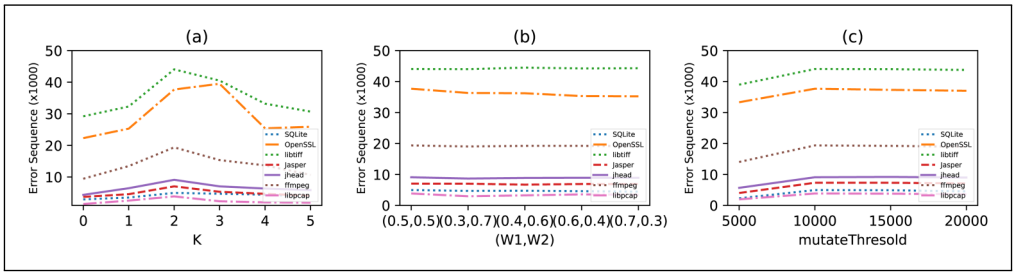

Figure 6. The influence of k, w1, w2, and mutateThreshold to error coverage.

metrics w1 and w2 in the cluster weight calculation algorithm (Algorithm 2), and the threshold mutateThreshold for the number of test cases generated by the fuzzer in the fuzzing-based iterative constraint solving algorithm (Algorithm 4). When investigating the impact of a specific parameter on the error coverage of HuntFUZZ, we maintain the values of the other variables at their defaults. We define the default values for these three parameters as follows: $k = 2$, $w1 = w2 = 0.5$, and mutateThreshold = 10,000.

5.3.1 Cluster distance k. In Algorithm 1, the parameter k signifies that the distance of error points within a cluster to their common parent node is less than or equal to k. Consequently, for a given tested program, the value of k influences the number of error points within a cluster. As depicted in Figure 6(a), we illustrate the impact of varying k values on the error coverage of HuntFUZZ across seven applications. Our results represent the error coverage observed after conducting tests on these applications for 24 h. When $k = 0$, it implies that the distance from the error point to the common parent is 0. In this scenario, the concolic executor needs to solve input constraints separately for each error point. As k increases, the number of error points within a cluster grows, allowing the concolic executor to only solve constraints for the paths that enable approaching or reaching the cluster. Consequently, the performance of the concolic executor improves. However, the concolic executor can only guide the fuzzer to the common parent node of these error points, requiring the fuzzer to still attempt coverage of error points within the cluster. Therefore, as k continues to increase, the performance of the fuzzer decreases. Hence, during the same testing duration, both excessively small and overly large values of k can adversely impact the error coverage of HuntFUZZ.

5.3.2 Cluster weights w1 and w2. In Algorithm 2, w1 and w2 are weights assigned to the parameters EPNum (number of injected error points) and clusterDistance (distance between the cluster and the current path) when calculating the cluster weight. These weights signify the importance of EPNum and clusterDistance in determining the weight of a cluster. In Figure 6(b), we present the impact of different values for w1 and w2 on the error coverage of HuntFUZZ. It can be observed that different values of w1 and w2 lead to slight variations in error coverage. In general, for the majority of applications, when $w1 = 0.5$ and $w2 = 0.5$, the error coverage of HuntFUZZ is maximized after 24 h.

5.3.3 mutateThreshold. In Algorithm 4, when the fuzzer attempts to cover a cluster of error points, the mutateThreshold signifies the point at which the exploration of this cluster stops once the fuzzer generates a specified number of test cases. In Figure 6(c), we document the impact of different mutateThreshold values on the error coverage of HuntFUZZ. It is evident that as the mutateThreshold value increases, for instance, from 5000 to 10,000, there is an improvement in HuntFUZZ's error coverage after 24 h. However, when the mutateThreshold value becomes excessively large (such as 20,000), the error coverage almost plateaus and may even exhibit a slight decline. This is attributed to the fact that an

excessively large mutateThreshold consumes too much time on that specific cluster, hindering the exploration of other error points and causing the overall error coverage to stabilize or slightly decrease.

5.4 Effectiveness of generated test cases

Although Sections 5.2 and 5.3 discuss the effectiveness of HuntFUZZ in terms of error coverage and code coverage, HuntFUZZ introduces a novel approach that sets it apart from existing SFI-based fuzzers by incorporating the concept of clustering error points. Therefore, we also need to validate HuntFUZZ's effectiveness in covering clusters and assess, at the cluster level, how many error points within clusters are reached. Additionally, in this section, we also address the following question: *based on the test cases output by the concolic executor, what percentage of the test cases generated by the fuzzer can reach the target cluster_max*, and *what percentage of the test cases are unable to reach any cluster?*

In practice, HuntFUZZ does not guarantee that all the error points within the cluster will be covered. Although we intuitively expect that fuzzer mutations of test cases generated by the concolic executor are likely to cover error points in the cluster (since error points are close to the common parent node and require fewer mutations to reach, as discussed in Section 2.2), this remains a probabilistic event. There may be cases where the fuzzer's mutated test cases do not cover the error points within the cluster. To address this, we conduct the following statistical experiments. For HuntFUZZ, we configure the parameter values associated with the algorithm at their default value, that is, $k = 2$, $w1 = w2 = 0.5$, and mutateThreshold = 10,000.

As shown in Table 3, in columns 2–4, we present, respectively, the total number of error points for each tested application, the total number of clusters, and the average number of error points per cluster. Column 5 records the average number of error points within each cluster that are covered by the test cases generated by HuntFUZZ. Column 6 records the average percentage of test cases generated by HuntFUZZ that are able to reach the target cluster (i.e. cluster_max in Algorithm 4). Specifically, within each iteration of the iterative constraint-solving process, we calculate the proportion of test cases that reach the cluster_max by dividing the number of such test cases by the total number of test cases generated during that iteration. The value in column 6 is the average of these proportions across all iterations. Column 7 records the percentage of test cases that fail to reach any cluster (including cluster_max and all other clusters). The numbers in columns 5–7 represent averages from five runs of HuntFUZZ.

From column 5 of Table 3, we can see that the test cases generated by HuntFUZZ cover a significant portion of the error points within clusters (an average of 88%). This high coverage of error points within clusters indicates the effectiveness of the test cases generated by HuntFUZZ, demonstrating that the concolic executor indeed helps the fuzzer explore most of the error points within the clusters. However, we also need to note that even though the concolic executor helps guide the fuzzer to the cluster, the percentage of fuzzer-generated test cases reaching a target cluster_max is not particularly high. This is because the fuzzer still randomly mutates the test cases generated by the concolic executor. That is why the percentage of the fuzzer-generated test cases reaching cluster_max is 39.9% (column 6). Additionally, 27.9% of the test cases generated by fuzzer do not reach any cluster at all (column 7).

6 Related work

Many recent studies[17–21] have utilized SFI-based fuzzing to trigger infrequently executed errors in programs, covering various scenarios such as USB drivers[17], device drivers[18], and IoT firmware.[20] These techniques typically mutate both error sequences and program inputs together, aiming to test whether error points will trigger error handling bugs. However, a common challenge in SFI-based fuzzing is the issue of early crash, where the execution stops if an error is encountered, preventing the testing from reaching deep error paths. To address this challenge, FIFUZZ[19] introduces a context-sensitive error injection method that effectively distinguishes shallow and deep error points, thus avoiding injecting shallow errors when testing deep error points. Similarly, iFIZZ[20] tackles the problem by saving the context of error points to prevent the reproduction of previously tested error points. On the other hand, EH-FUZZ[21] argues that using code coverage to guide error sequence generation is insufficient since if two test cases trigger the same error point but in different execution contexts, these methods would consider them as equivalent. However, the contexts in which these error points are triggered may differ, and code coverage cannot reflect the context information of error points. In light of this, EH-FUZZ proposes using error coverage to guide the generation of error sequences. This approach allows for a more comprehensive testing of handling errors by considering the diverse contexts in which errors can occur, rather than relying solely on code coverage-based guidance.

However, existing SFI-based fuzzing approaches rely on traditional fuzzing for test case generation, these approaches do not consider the correlation of paths where error points are located. This leads to fuzzers needing to repeatedly generate test cases to explore duplicated paths, thereby diminishing the efficiency of the fuzzer. This paper introduces HuntFUZZ,

Table 3. The coverage of error points within clusters and the percentage of the test cases generated by the fuzzer that can (or cannot) reach (any) cluster.

Tested program	Number of error points	Number of clusters (k = , 2)	Average number of error points in clusters	Average number of error points covered in cluster	Test cases that reach $\text{cluster}_{\text{max}}$	Test cases that fail to reach any cluster
exiv2	70	17	4.1	3.7	39.4%	32.5%
gdk-pixbuf-pixdata	63	39	1.6	1.0	39.5%	28.7%
jasper	92	22	4.2	4.0	38.5%	24.7%
jhead	359	188	1.9	1.6	38.4%	29.6%
libtiff	695	182	3.8	3.2	44.6%	31.6%
lame	332	73	4.5	3.9	44.6%	24.7%
mp3gain	198	38	5.2	4.8	39.7%	29.9%
swftools	571	150	3.8	3.6	35.3%	25.9%
ffmpeg	112	31	3.6	3.0	37.2%	22.3%
flvmeta	254	56	4.5	4.2	42.2%	30.7%
Bento4	348	69	5.0	4.6	41.4%	32.5%
cflow	88	51	1.7	1.5	41.0%	28.5%
ncurses	210	46	4.6	4.1	35.1%	33.8%
jq	485	186	2.6	2.4	43.2%	24.5%
mujs	279	73	3.8	3.4	42.6%	32.9%
pdftotext	165	45	3.7	3.2	40.0%	24.9%
SQLite	91	21	4.3	3.7	43.5%	32.6%
binutils	144	57	2.5	2.0	43.5%	26.0%
libpcap	329	70	4.7	4.2	37.9%	29.2%
tcpdump	626	223	2.8	2.4	38.3%	27.5%
vim	270	108	2.5	2.2	43.8%	24.2%
bison	125	33	3.8	3.3	38.7%	29.4%
nasm	26	5	5.2	4.7	35.9%	32.1%
catdoc	69	14	4.9	4.7	41.5%	30.9%
clamav	1247	277	4.5	4.2	44.9%	24.1%
gif2png+libpng	65	17	3.8	3.4	35.9%	29.7%
openssl	102	29	3.5	3.1	36.6%	30.6%
btrfs	351	184	1.9	1.7	40.7%	20.9%
xfs	171	34	5.0	4.6	37.7%	20.0%
jfs	100	25	4.0	3.8	36.3%	32.3%
cephfs	140	36	3.9	3.6	37.4%	30.3%
xhci	104	38	2.7	2.0	41.2%	30.4%
vmxnet3	43	26	1.7	1.4	44.0%	26.9%
man-db	158	63	2.5	2.4	43.8%	31.3%
woff2	139	55	2.5	2.2	35.8%	25.7%
gzip	272	181	1.5	1.2	37.4%	25.0%
bzip2	365	110	3.3	2.8	42.9%	25.7%
sassc	284	94	3.0	2.4	35.8%	27.5%
tidy	381	97	3.9	3.4	35.5%	23.6%
jqlang	67	27	2.5	2.3	44.0%	24.5%
bash	351	85	4.1	4.0	42.5%	29.4%
mksh	343	127	2.7	2.6	36.6%	22.4%
Average	254.4	78.6	3.5	3.1 (88%)	39.9%	27.9%

which addresses the aforementioned limitations in SFI-based fuzzing by incorporating concolic execution. Taking into account the correlation among paths where certain error points are situated, HuntFUZZ solves constraints for paths that enable approaching or reaching the cluster of certain error points, thereby enhancing the efficiency of the fuzzer in exploring these error points.

7 Conclusion and future work

In this paper, we introduce HuntFUZZ, by considering correlations among paths containing error points and selectively solving constraints for paths that enable approaching or reaching error points with path overlaps. Specifically, we propose

an algorithm for clustering error points with common paths, calculating the weight of each cluster, and utilizing an optimization strategy to explore clusters with the highest weights. HuntFUZZ surpasses current SFI-based fuzzing methods with faster and superior error coverage, specifically showing a substantial 38.9% increase in error coverage compared to the most advanced SFI-based fuzzing method. Moreover, HuntFUZZ detects zero-day bugs that other tools failed to find.

Furthermore, although we observe the correlation of error points' paths, such path correlations may be prevalent across the fuzzer's targets beyond error handling scenarios. In addition to error-handling scenarios, more general contexts may also benefit from clustering targets to reduce fuzzers' exploration of redundant paths. We will delve deeper into this in future work.

In addition, we still need to manually verify whether the error points identified by HuntFUZZ are actual errors, as demonstrated in Section 5. We aim to automate this process in our future work. Typical error point patterns often include recurring function calls, memory allocation issues, and boundary condition checks that can be systematically identified. Potential automation could involve developing tools to recognize these common patterns using static analysis. By focusing on frequently occurring patterns, automation can streamline the identification process, making it more manageable despite the high volume of potential error sites. This will be a focus of our future work.

ORCID iD

Ping Chen https://orcid.org/0000-0002-8517-0580

Funding

The authors disclosed receipt of the following financial support for the research, authorship, and/or publication of this article: This work was supported by National Key R&D Program of China under grant No. 2022YFB3102902.

Declaration of conflicting interests

The authors declared no potential conflicts of interest with respect to the research, authorship, and/or publication of this article.

References

1. Gunawi HS, Rubio-González C, Arpaci-Dusseau AC, et al. Eio: Error handling is occasionally correct. In: *6th USENIX Conference on File and Storage Technologies, FAST'08, San Jose, California, February 26–29*, 2008, USENIX Association, 2560 Ninth St. Suite 215 Berkeley, CA United States.
2. Saha S, Lozi JP, Thomas G, et al. Hector: Detecting resource-release omission faults in error-handling code for systems software. In: *2013 43rd annual IEEE/IFIP international conference on dependable systems and networks (DSN)*, Budapest, Hungary, 24–27 June 2013, pp.1–12. IEEE.
3. Weimer W and Necula GC. Finding and preventing run-time error handling mistakes. In: *Proceedings of the 19th annual ACM SIGPLAN conference on object-oriented programming, systems, languages, and applications*s (OOPSLA '04), Vancouver, British Columbia, Canada, Oct. 2004, pp.419–431, Association for Computing Machinery, New York, United States.
4. Cabral B and Marques P. Exception handling: a field study in java and. net. In: *ECOOP 2007–object-oriented programming: 21st European conference, Berlin, Germany, July 30–August 3, 2007. Proceedings 21*, 2007, pp.151–175. Springer, Berlin, Heidelberg.
5. Ebert F and Castor F. A study on developers' perceptions about exception handling bugs. In: *2013 IEEE international conference on software maintenance*, *Eindhoven, Netherlands, 22–28 September* 2013, Eindhoven, Netherlands, pp.448–451. IEEE, NW Washington, DC, United States.
6. Kery MB, Le Goues C and Myers BA. Examining programmer practices for locally handling exceptions. In: *Proceedings of the 13th international conference on mining software repositories (MSR), Austin Texas, May 14–22*, 2016, pp.484–487, Association for Computing Machinery, New York, United States.
7. Shah H, Görg C and Harrold MJ. Why do developers neglect exception handling? In: *Proceedings of the 4th international workshop on Exception handling (WEN),* Atlanta Georgia, November 14, 2008, pp.62–68, Association for Computing Machinery, New York, United States.
8. Fu C, Ryder BG, Milanova A, et al. Testing of Java web services for robustness. In: *Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis, Boston, Massachusetts, USA, July*, 2004, pp.23–34, Association for Computing Machinery, New York, United States.
9. Askarov A and Sabelfeld A. Catch me if you can: permissive yet secure error handling. In: *Proceedings of the ACM SIGPLAN fourth workshop on programming languages and analysis for security, Dublin Ireland, June 15–21*, 2009, pp.45–57, Association for Computing Machinery, New York, United States.
10. Jana S, Kang YJ, Roth S, et al. Automatically detecting error handling bugs using error specifications. In: *25th USENIX security symposium (USENIX Security 16), Austin, TX, Aug.* 2016, pp.345–362, USENIX Association, 2560 Ninth St. Suite 215 Berkeley, CA, United States.

11. Lawall J, Laurie B, Hansen RR, et al. Finding error handling bugs in OpenSSL using Coccinelle. In: *2010 European dependable computing conference, Valencia, Spain, 28–30 April*, 2010, pp.191–196. IEEE.
12. Zuo C, Wu J and Guo S. Automatically detecting ssl error-handling vulnerabilities in hybrid mobile web apps. In: *Proceedings of the 10th ACM symposium on information, computer and communications security, Singapore, Republic of Singapore, April 14–17*, 2015, pp.591–596, Association for Computing Machinery, New York, United States.
13. M. CORPORATION. Cve-2019-7846, 2019. https://nvd.nist.gov/vuln/detail/CVE-2019-7846.
14. M. CORPORATION. Cve-2019-2240, 2019. https://nvd.nist.gov/vuln/detail/CVE-2019-2240.
15. M. CORPORATION. Cve-2019-1750, 2019. https://nvd.nist.gov/vuln/detail/CVE-2019-1750.
16. M. CORPORATION. Cve-2019-1785, 2019. https://nvd.nist.gov/vuln/detail/CVE-2019-1785.
17. Patrick-Evans J, Cavallaro L and Kinder J. {*POTUS*}: Probing {*Off − The − Shelf*} {*USB*} drivers with symbolic fault injection. In: *11th USENIX workshop on offensive technologies (WOOT 17), Vancouver BC Canada, August 14–15*, 2017, USENIX Association, 2560 Ninth St. Suite 215 Berkeley, CA, United States.
18. Jiang ZM, Bai JJ, Lawall J, et al. Fuzzing error handling code in device drivers based on software fault injection. In: *software reliability 2019 IEEE 30th International symposium on engineering (ISSRE), Berlin, Germany, 27–30 Oct*, 2019, pp.128–138. IEEE.
19. Jiang ZM, Bai JJ, Lu K, et al. Fuzzing error handling code using {Context-Sensitive} software fault injection. In: *29th USENIX security symposium (USENIX Security 20), August 12–14*, 2020, pp.2595–2612, USENIX Association, 2560 Ninth St. Suite 215 Berkeley, CA, United States.
20. Liu P, Ji S, Zhang X, et al. Ifizz: Deep-state and efficient fault-scenario generation to test IoT firmware. In: *2021 36th IEEE/ACM International conference on automated software engineering (ASE), Melbourne Australia, November 15–19*, 2021, pp.805–816. IEEE.
21. Bai JJ, Fu ZX, Xie KT, et al. Testing error handling code with software fault injection and error-coverage-guided fuzzing. *IEEE Trans Dependable Secure Comput* 2023, vol. 21, pp.1724–1739.
22. Rosenberg HA and Shin KG. Software fault injection and its application in distributed systems. In: *FTCS-23 the twenty-third international symposium on fault-tolerant computing, Toulouse, France, June 22–24*, 1993, pp.208–217. IEEE.
23. Yang X, Chen Y, Eide E, et al. Finding and understanding bugs in c compilers. In: *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation, San Jose, California USA, June 4–8*, 2011, pp.283–294, Association for Computing Machinery, New York, United States.
24. Chen Y, Groce A, Zhang C, et al. Taming compiler fuzzers. In: *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation, Seattle, Washington, USA, June 16–19*, 2013, pp.197–208, Association for Computing Machinery, New York, United States.
25. Godefroid P, Kiezun A and Levin MY. Grammar-based whitebox fuzzing. In: *Proceedings of the 29th ACM SIGPLAN conference on programming language design and implementation, Tucson, AZ, USA, June 7–13*, 2008, pp.206–215, Association for Computing Machinery, New York, United States.
26. Wang J, Chen B, Wei L, et al. Skyfire: Data-driven seed generation for fuzzing. In: *2017 IEEE symposium on security and privacy (SP), San Jose, CA, USA, May 22–26*, 2017, pp.579–594. IEEE.
27. Rawat S, Jain V, Kumar A, et al. Vuzzer: Application-aware evolutionary fuzzing. In: *NDSS*, Vol. 17, 2017, pp.1–14.
28. Holler C, Herzig K and Zeller A. Fuzzing with code fragments. In: *21st USENIX Security Symposium (USENIX Security 12), Bellevue, WA, August 8–10*, 2012, pp.445–458, USENIX Association, 2560 Ninth St. Suite 215 Berkeley, CA, United States.
29. 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, Montpellier, France, September 3–7*, 2018, pp.475–485, Association for Computing Machinery, New York, United States.
30. Pham VT, Böhme M, Santosa AE, et al. Smart greybox fuzzing. *IEEE Trans Softw Eng* 2019; 47: 1980–1997.
31. Aschermann C, Frassetto T, Holz T, et al. Nautilus: fishing for deep bugs with grammars. In: *NDSS*, 2019.
32. Padhye R, Lemieux C, Sen K, et al. Semantic fuzzing with zest. In: *Proceedings of the 28th ACM SIGSOFT international symposium on software testing and analysis, Beijing, China, July 15–19*, 2019, pp.329–340, Association for Computing Machinery, New York, United States.
33. Wang J, Chen B, Wei L, et al. Superion: Grammar-aware greybox fuzzing. In: *2019 IEEE/ACM 41st international conference on software engineering (ICSE), Montreal, Quebec, Canada, 27 May* 2019, pp.724–735. IEEE.
34. Zalewski M. American fuzzy lop, 2023. https://github.com/google/AFL.
35. Google. Honggfuzz, 2023. https://google.github.io/honggfuzz/.
36. Böhme M, Pham VT and Roychoudhury A. Coverage-based greybox fuzzing as Markov chain. In: *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security, Vienna, Austria, October 24–28*, 2016, pp.1032–1043, Association for Computing Machinery, New York, United States.

37. Gan S, Zhang C, Qin X, et al. Collafl: Path sensitive fuzzing. In: *2018 IEEE symposium on security and privacy (SP), San Francisco, CA, May 21–23*, 2018, pp.679–696. IEEE.
38. Yun I, Lee S, Xu M, et al. Qsym: a practical concolic execution engine tailored for hybrid fuzzing. In: *Proceedings of the 27th USENIX security symposium (USENIX Security 18), Baltimore MD, USA, August 15–17*, 2018, pp.745–761, USENIX Association, 2560 Ninth St. Suite 215 Berkeley, CA, United States.
39. Aschermann C, Schumilo S, Blazytko T, et al. Redqueen: Fuzzing with input-to-state correspondence. In: *Proceedings of the 26th Annual network and distributed system security symposium (NDSS), vol. 19, San Diego, CA, February 24–27*, 2019, pp.1–15.
40. M. CORPORATION. Cve-2019-4332, 2019. https://nvd.nist.gov/vuln/detail/CVE-2019-4332.
41. Li Y, Ji S, Chen Y, et al. {UNIFUZZ}: A holistic and pragmatic {Metrics-Driven} platform for evaluating fuzzers. In: *30th USENIX security symposium (USENIX Security 21), Vancouver, B.C., Canada, August 11–13*, 2021, pp.2777–2794.
42. Fioraldi A, Maier D, Eißfeldt H, et al. {AFL++}: Combining incremental steps of fuzzing research. In: *14th USENIX workshop on offensive technologies (WOOT 20), 11 August*, 2020, pp.10–21, USENIX Association, 2560 Ninth St. Suite 215 Berkeley, CA, United States.
43. Böhme M, et al. Directed greybox fuzzing. In: *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security, Dallas, Texas, USA, 30 October–3 November 2017*, Association for Computing Machinery, New York, United States.
44. Bai JJ, Wang YP, Liu HQ, et al. Mining and checking paired functions in device drivers using characteristic fault injection. *Inf Softw Technol* 2016; 73: 122–133.
45. Banabic R and Candea G. Fast black-box testing of system recovery code. In: *Proceedings of the 7th ACM European conference on computer systems, Bern, Switzerland, April 10–13*, 2012, pp.281–294, Association for Computing Machinery, New York, United States.
46. Cong K, Lei L, Yang Z, et al. Automatic fault injection for driver robustness testing. In: *Proceedings of the 2015 international symposium on software testing and analysis, Baltimore, MD, USA, July 13–17*, 2015, pp.361–372, Association for Computing Machinery, New York, United States.
47. Marinescu PD and Candea G. Lfi: a practical and general library-level fault injector. In: *2009 IEEE/IFIP International Conference on Dependable Systems & Networks, storil/Lisbon, Portugal*, 2009, pp.379–388. IEEE.
48. Mendonca M and Neves N. Robustness testing of the Windows DDK. In: *37th annual IEEE/IFIP international conference on dependable systems and networks (DSN'07), June 25–28*, 2007, pp.554–564. IEEE, NW Washington, DC, United States.
49. Susskraut M and Fetzer C. Automatically finding and patching bad error handling. In: *2006 Sixth European dependable computing conference, Coimbra, Oct. 18–20*, 2006, pp.13–22. IEEE.
50. Zhang P and Elbaum S. Amplifying tests to validate exception handling code. In: *2012 34th International conference on software engineering (ICSE), Zurich, Switzerland, June 2–9*, 2012, pp.595–605. IEEE.
51. Fuzzing IW. Sage: Whitebox fuzzing for security testing. *SAGE* 2012; 10:1, pp.20–27.
52. Stephens N, Grosen J, Salls C, et al. Driller: augmenting fuzzing through selective symbolic execution. In: *NDSS*, Vol. 16, 2016, pp.1–16.
53. Cadar C, Dunbar D, Engler DR, et al. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: *OSDI*, vol. 8, 2008, pp.209–224, USENIX Association, 2560 Ninth St. Suite 215 Berkeley, CA, United States.
54. Cha SK, Avgerinos T, Rebert A, et al. Unleashing mayhem on binary code. In: *2012 IEEE symposium on security and privacy, San Francisco, CA, USA, May 20–23*, 2012, pp.380–394. IEEE.
55. Shoshitaishvili Y, Wang R, Salls C, et al. Sok:(state of) the art of war: Offensive techniques in binary analysis. In: *2016 IEEE symposium on security and privacy (SP), San Jose, CA, USA, May 22–26*, 2016, pp.138-157. IEEE.
56. Pak BS. Hybrid fuzz testing: discovering software bugs via fuzzing and symbolic execution. *Master's thesis, School Comput Sci Carneg Mellon Univ* 2012.
57. Majumdar R and Sen K. Hybrid concolic testing. In: *29th international conference on software engineering (ICSE'07), Minneapolis, MN, May 20–26*, 2007, pp.416–426. IEEE.
58. Poeplau S and Francillon A. Symbolic execution with {SymCC}: Don't interpret, compile! In: *29th USENIX security symposium (USENIX security 20), August 12–14*, 2020, pp.181–198, USENIX Association, 2560 Ninth St. Suite 215 Berkeley, CA, United States.
59. Python. https://www.python.org/.
60. Z3 time out issue. https://github.com/Z3Prover/z3/issues/419.
61. Jiang ZM, Bai JJ, Lawall J, et al. Fuzzing error handling code in device drivers based on software fault injection. In: *2019 IEEE 30th international symposium on software reliability engineering (ISSRE), Berlin, Germany, Oct. 28–31*, 2019, pp.128–138. IEEE.
62. LLVM pass. https://llvm.org/docs/WritingAnLLVMPass.html.
63. Serebryany K, Bruening D, Potapenko A, et al. {AddressSanitizer}: A fast address sanity checker. In: *2012 USENIX annual technical conference (USENIX ATC 12), Boston, MA, June 13–1*, 2012, pp.309–318, USENIX Association, 2560 Ninth St. Suite 215 Berkeley, CA, United States.

64. Msan: memory sanitizer, 2019. http://github.com/google/sanitizers/wiki/MemorySanitizer.
65. Luk CK, Cohn R, Muth R, et al. Pin: building customized program analysis tools with dynamic instrumentation. *ACM SIGPLAN Notices* 2005; 40: 190–200.
66. Research M. https://github.com/Z3Prover/z3.
67. man db. https://gitlab.com/man-db/man-db/-/tree/2.12.0?ref_type=tags.
68. woff2. https://github.com/google/woff2.
69. gzip. https://ftp.gnu.org/gnu/gzip/.
70. bzip2. https://github.com/vim/vim/tree/v9.0.1343.
71. sassc. https://github.com/sass/sassc.
72. tidy. https://github.com/htacg/tidy-html5.
73. jqlang. https://github.com/jqlang/jq.
74. bash. https://ftp.gnu.org/gnu/bash/bash-5.2.21.tar.gz.
75. mksh. http://www.mirbsd.org/MirOS/dist/mir/mksh/mksh-R59c.tgz.
76. K S. OSS-Fuzz-google's continuous fuzzing service for open source software.

Appendix A. Other zero-day bugs found by HuntFUZZ

In this section, we present zero-day bugs discovered by HuntFUZZ in other applications.

READ memory access bug in tidy. As shown in Code 6, the InsertDocType function contains a while loop with the condition !nodeIsHTML(element) (line 6). The expanded nodeIsHTML macro checks if element and element->tag exist, and if element->tag->id equals a specific value tid (lines 1 and 2). The bug occurs when element is NULL. In this case, the condition !nodeIsHTML(element) evaluates to true, leading to the loop's execution. Inside the loop body, there is an attempt to access a NULL pointer via element->parent (line 7), resulting in a "READ memory access" error due to invalid memory access.

```
1 #define TagIsId(node, tid) ((node) && (node)->tag && (node)->tag->id == tid)
2 #define nodeIsHTML( node ) TagIsId( node, tidyTag_HTML )
3
4 static void InsertDocType( tidyDocImpl* doc, Node *element, Node *doctype )
5 {   ...
6     while ( !nodeIsHTML(element) )
7         element = element->parent;
8     ...
9 }
```

Code 6 READ memory access bug in `tidy`.

Heap overflow bug in jqlang. As shown in Code 7, the function jvp_literal_number_literal calculates a length len as jvp_dec_number_ptr(n)->digits + 14 (line 4). It then allocates a memory buffer of size len and stores the pointer in plit->literal_data using jv_mem_alloc (line 5). Next, the function decNumberToString is called to convert pdec into a string and store it in the allocated buffer (line 6). However, decNumberToString calculates the string length as len + 15, which is larger than the allocated buffer size. Since the buffer is only allocated with length len, this can cause decNumberToString to write beyond the allocated memory, leading to a heap overflow.

```
1 static const char* jvp_literal_number_literal(jv n)
2 {   ...
3     if (plit->literal_data == NULL) {
4     int len = jvp_dec_number_ptr(n)->digits {\xmlplus} 14;
5     plit->literal_data = jv_mem_alloc(len);
6     decNumberToString(pdec, plit->literal_data);
7     }
8     ...
9 }
```

Code 7 Heap overflow bug in `jqlang`.

Segmentation fault in bash and mksh. During testing of both bash and mksh, we encounter segmentation fault errors. In bash, the issue occurred in parse.y within the function pop_string, where accessing t->expander->flags &= AL_BEINGEXPANDED (Code 8, line 21) failed due to inaccessible addresses set for variables t->expand and t->next. Similarly, in mksh, a segmentation fault occurred in the function wdscan (Code 9, line 7) because its parameter wp pointed to an invalid address. The reasons behind these issues in both applications are currently unclear. We have provided proof-of-concept exploits for both bugs to the developers and are awaiting their responses.

```
static void
pop_string ()
{
  STRING_SAVER *t;
  FREE (shell_input_line);
  shell_input_line = pushed_string_list->saved_line;
  shell_input_line_index = pushed_string_list->saved_line_index;
  shell_input_line_size = pushed_string_list->saved_line_size;
  shell_input_line_len = pushed_string_list->saved_line_len;
  shell_input_line_terminator = pushed_string_list->saved_line_terminator;
#if defined (ALIAS)
  if (pushed_string_list->expand_alias)
    parser_state |= PST_ALEXPNEXT;
  else
    parser_state &= ~PST_ALEXPNEXT;
#endif
  t = pushed_string_list;
  pushed_string_list = pushed_string_list->next;
#if defined (ALIAS)
  if (t->expander)
    t->expander->flags &= ~AL_BEINGEXPANDED;
#endif
  free ((char *)t);
  set_line_mbstate ();
}
```

Code 8 Segmentation fault in bash.

```
const char *
wdscan(const char *wp, int c)
{
    int nest = 0;
    while (/* CONSTCOND */ 1)
        switch (*wp++) {
        case EOS:
            return (wp);
        case ADELIM:
            if (c == ADELIM &{\xmlamp} nest == 0)
                return (wp + 1);
            if (ord(*wp) == ORD(/*{*/ '}'))
                goto wdscan_csubst;
            /* FALLTHROUGH */
    ...
    }
}
```

Code 9 Segmentation fault in mksh.