PATA: Fuzzing with Path Aware Taint Analysis

Jie Liang*, Mingzhe Wang*, Chijin Zhou*, Zhiyong Wu*, Yu Jiang*✉, Jianzhong Liu*, Zhe Liu†, and Jiaguang Sun*
*School of Software, Tsinghua University, KLISS, BNRist, Beijing, China
†Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, Nanjing, China

2022 IEEE Symposium on Security and Privacy (SP) | 978-1-6654-1316-9/22/$31.00 ©2022 IEEE | DOI: 10.1109/SP46214.2022.9833594

***Abstract*—Taint analysis assists fuzzers in solving complex fuzzing constraints by inferring the influencing input bytes. Execution paths in real-world programs often reach loops, where constraints in these loops can be visited and recorded multiple times. Conventional taint analysis techniques experience difficulties when distinguishing between multiple occurrences of the same constraint. In this paper, we propose PATA, a fuzzer that implements path-aware taint analysis, i.e. one that distinguishes between multiple occurrences of the same variable based on the execution path information. PATA does so using the following steps. First, PATA identifies variables used in constraints and constructs the Representative Variable Sequence (RVS), consisting of occurrences of all representative constraint variables and their values. Next, PATA perturbs the input, matches its RVS with that of the original input, and looks for value changes to identify the influencing input bytes for each entry in the RVS. Finally, PATA mutates the corresponding input bytes to solve constraints in the given path.**

To demonstrate the effectiveness of PATA over conventional taint analysis methods, we evaluated its performance on the benchmarks Google's fuzzer-test-suite and LAVA-M against AFL, MOpt, TortoriseFuzz, VUzzer, Angora, REDQUEEN, and GREYONE. On Google's fuzzer-test-suite, PATA outperformed these state-of-the-art fuzzers by 29%–1830% and 7%–87% in the number of unique paths found and basic blocks covered, respectively. More importantly, it found more bugs than the comparison fuzzers, including 17 unlisted ones. On LAVA-M, PATA performed the best out of all evaluated fuzzers and found 2602 bugs. On open-source projects, PATA found 40 previously unknown bugs, with 12 of them confirmed as CVEs.

I. INTRODUCTION

Mutation-based greybox fuzzing is one of the most popular techniques for mining vulnerabilities [1, 2, 5, 8, 14, 15, 25, 29, 30]. Most mutation-based fuzzers employ an evolutionary algorithm, which prioritizes inputs that trigger new program coverage [3, 7, 20, 24, 36]. While the algorithm can determine the quality of an existing input, mutated inputs are still generated using random techniques. This blind generation scheme severely limits the overall performance of such fuzzers.

Recently, taint analysis has gained much interest in assisting fuzzing. Taint analysis improves the mutation quality of fuzzers by identifying the influencing input bytes of a given constraint. This allows the fuzzer to focus on these *critical bytes* instead of touching all bytes universally. As a result, fuzzers can explore program logic more efficiently. There are two mainstream taint analysis techniques, namely *propagation* and *inference*. *Propagation-based taint analysis* [4, 11, 26, 32] taints each byte of the input with different labels, then propagates these labels using propagation rules during program execution. *Inference-based taint analysis* [6, 17, 22, 34] perturbs the input bytes repeatedly and collects the variables' values during program execution. If a variable's value changes, then there are data dependencies between the perturbed bytes and this variable. State-of-the-art fuzzers can achieve significantly better performance and detect many bugs due to taint analysis.

However, *over-tainting* and *under-tainting* will incur major penalties in analysis correctness, and consequently, fuzzing performance. One of the most prominent causes is *path-unawareness*: loops and multiple function call sites may result in a given constraint being visited multiple times and influenced by different input bytes in a single execution path. Despite having well-known control-flow paradigms when analyzing real-world programs, both aforementioned taint analysis approaches are error-prone in such scenarios. (1) *Propagation-based taint analysis* suffers from over-tainting without path-awareness. It fails to distinguish between different occurrences and cannot determine when to clear redundant labels. Therefore, when the same constraint is visited multiple times, its label continues to accumulate, resulting in over-tainting. In the worst case, all input bytes are labeled as critical to one constraint, which is no help to mutation at all. (2) *Inference-based taint analysis* suffers from under-tainting without path-awareness. First, the byte-level mutation used by this approach may alter the execution path, resulting in visiting a constraint for a different amount of times or skipping it altogether. Its lack of path-awareness may lead to incorrect or incomplete results. Second, even if the path remains the same across mutations, the path-unaware inference technique can only capture each constraint's value once; therefore, the bytes which affect other occurrences are missed in the results.

In this paper, we propose a novel path-aware approach for *inference-based* taint analysis. Our approach does so using the following steps.

First, it collects constraint variables that represent program states along an execution path into a Representative Variable Sequence. *The challenge here is to trace representative variables reasonably in taint analysis.* Because the analysis is based on variable occurrences, the highest fidelity approach is to record the states of all variables during execution. However, the cost to record them is prohibitive, thus sampling must be used instead. Therefore, a reasonable strategy must be deployed to trace a subset of all variables, the *representative variables*, without compromising analysis precision.

Then, it identifies corresponding *critical bytes* for each variable occurrence. During the inference process, the fuzzer perturbs each byte of an input and monitors the value changes of variable occurrences to identify *critical bytes*. However, the path might deviate from the original path. For example, a

✉Yu Jiang and Zhe Liu are the corresponding authors.

constraint variable might completely disappear or still appear but with different occurrence counts. *The challenge here is to match variable occurrences correctly when the execution path change.* Therefore, a matching algorithm is required to correctly match variable occurrences between changed paths.

Finally, it leverages the results of path-aware taint analysis to guide the fuzzer's mutation process. *The challenge here is to utilize analysis results effectively to explore other branches along the original path.* During execution, the results of path-aware taint inference consist of the critical bytes and values of each variable occurrence. If the results are sufficiently utilized, the fuzzer's mutation effectiveness can be improved, allowing the fuzzer to solve magic bytes checks as well as other complicated constraints. Thus a mutation strategy is needed to exploit these results sufficiently.

To address these challenges, we propose PATA, a fuzzer that implements the aforementioned procedures. ① PATA collects constraint variables along paths, by narrowing down the scope to variables with high impacts to constraints through searching and increasing the sensitivity to input bytes by backtracking. ② It then identifies critical bytes by extracting the subsequence for a given variable from the path, matching the variables and comparing the values before and after perturbation. ③ Finally, PATA mutates critical bytes by employing a path-oriented mutation to exploit the analyzed results effectively. PATA tries to explore every uncovered branch in a given execution path. With the values and features of variables, PATA selects proper mutation methods on critical bytes to bypass constraints.

We evaluated PATA on two widely used benchmarks: Google's fuzzer-test-suite, which contains a series of real-world programs, and LAVA-M, which consists of four real-world programs with synthetic bugs inserted. The results show that PATA demonstrates excellent performance. On Google's fuzzer-test-suite, PATA outperformed seven state-of-the-art fuzzers (e.g. Angora and GREYONE) by 29%–1830% and 7%–87% in the number of unique paths found and basic blocks covered. In addition, PATA was capable of finding more bugs than others, including 17 unlisted ones [1] . On LAVA-M, PATA performed best and found 2602 bugs. We further evaluated PATA on a diverse set of open-source projects. PATA found 40 new unknown bugs, with 12 confirmed as CVEs.

II. MOTIVATING EXAMPLE

Problem Description. A PNG file consists of a series of chunks. Figure 1 shows the first two chunks of a simple PNG file: the image header chunk, IHDR; and the image data chunk, IDAT. Listing 1 is extracted from libpng. It uses a loop to determine the type of the chunk. Specifically, libpng first gets the chunk length and the chunk name (e.g. "IHDR"). Then it compares the chunk name with predefined values sequentially until the matched type is found and handles them with their corresponding logic. Let $v1$ and $v2$ represent the variables at Line 7 and Line 13. For the sample input, as shown in Figure 2, its path is "$v1 \rightarrow v2 \rightarrow v1 \rightarrow v2$". Note that both constraint variables are visited exactly two times. Due to the inability to distinguish between a variable's different

[1]https://github.com/PATA-FUZZ/pata

```
 1  void PNGAPI png_read_info(structp png_ptr, /*...*/) {
 2    // ...
 3    for (/*...*/) {
 4      png_uint_32 length = png_read_chunk_header(png_ptr);
 5      png_bytep chunk_name = png_ptr->chunk_name;
 6      // v1 {lhs: chunk_name, rhs: png_IDAT}
 7      if (chunk_name == png_IDAT) {
 8        if ((png_ptr->mode & PNG_HAVE_IHDR) == 0)
 9         chunk_error(png_ptr,"Missing␣IHDR␣before␣IDAT");
10         //...
11      }
12      // v2 {lhs: chunk_name, rhs: png_IHDR}
13      if (chunk_name == png_IHDR) {
14        // ... png_handle_IHDR ...
15        png_ptr->mode |= PNG_HAVE_IHDR;//...
16      } // ...
17    }
18  }
```

Listing 1. Code snippet extracted from libpng. This example illustrates the necessity of path-awareness in taint analysis. When the *for* loop is executed multiple times, the constraint variables v_1 and v_2 have different influencing input bytes on different iterations. Path-unaware taint analysis methods cannot perceive this information, leaving fuzzers with inaccurate information regarding constraints.

occurrences, propagation-based taint analysis will over-taint while inference-based taint analysis will under-taint.

```
        0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
000000 89  P  N  G 0d 0a 1a 0a 00 00 00 0d  I  H  D  R
000010 00 00 00 20 00 00 00 20 08 03 00 00 00 44 a4 8a
000020 3c 00 00 00 61  I  D  A  T 78 da dc 93 31 0e c0
  ...                          ...
```

Fig. 1. A sample PNG file. It shows the first two necessary chunks: the image header chunk, IHDR; and the image data chunk, IDAT.

Propagation-Based Path-Unaware Taint Analysis. This approach taints the input bytes with labels and propagates them during program execution. The path-unawareness causes it to over-taint the critical bytes for constraints. Specifically, following the execution path and propagation rules, when dealing with the first chunk, the labels of bytes at 0x0c–0x0f ("IHDR") propagate to $v1$. But when parsing the second chunk, the labels of bytes at 0x25–0x28 ("IDAT") also propagate to $v1$. Because the labels for the first occurrence are not cleaned up, $v1$ has the labels of all two chunk names. In other words, the bytes at 0x0c–0x0f and 0x25–0x28 will all be considered as its critical bytes. However, the bytes at 0x0c–0x0f only affect the first occurrence of $v1$, and the bytes at 0x25–0x28 only affect the second occurrence. In more common cases, more consecutive bytes would be considered as critical bytes. The results make it difficult for fuzzers to choose appropriate bytes to mutate.

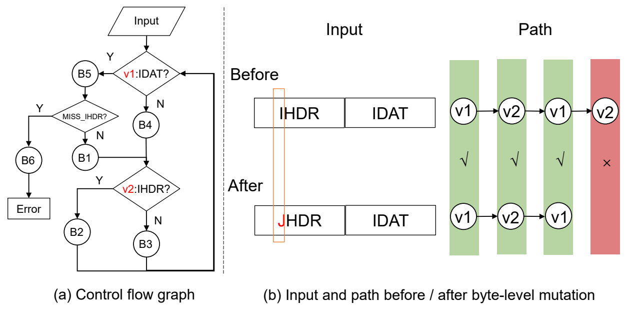

Fig. 2. When execution paths alter after input perturbation, PATA utilizes a matching algorithm to determine which constraint variable occurrence after perturbation matches with a constraint variable occurrence in the original path. Matched pairs are marked with ✓in the figure.

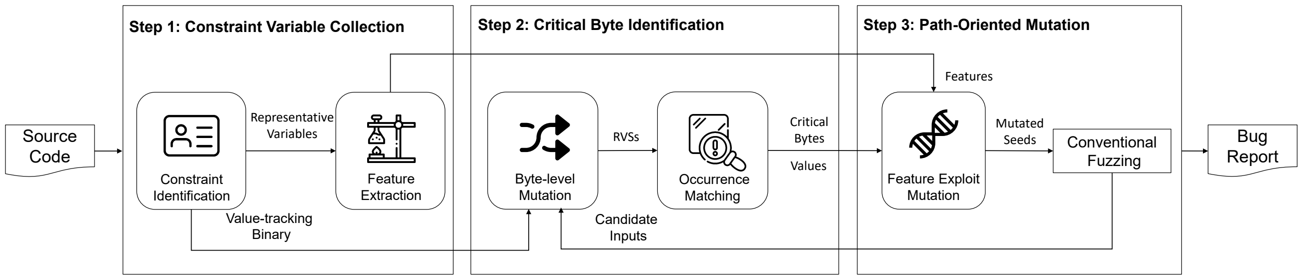

Fig. 3. Design of PATA. 1) In Step 1, PATA collects constraint variables by identifying representative constraint variables and instrumenting the program to collect paths and construct the Representative Variable Sequence. Each variable's relevant features are also extracted. 2) In Step 2, PATA identifies critical bytes by byte-level mutation and occurrence matching. In byte-level mutation, it collects the RVSs consisting of variable occurrences on different runs. PATA matches the RVSs of perturbed inputs with that of the original input to identify corresponding critical bytes for each variable occurrence. 3) In Step 3, PATA performs path-oriented mutation. At each entry along the RVS, using the critical bytes for each occurrence, combined with values and detailed features, PATA selects proper mutation methods to bypass the constraints. Finally, the mutated seeds are passed to conventional fuzzing to explore the program states and find new bugs.

Inference-Based Path-Unaware Taint Analysis. The conventional approach infers taint information by mutating each byte and checking whether the value changes. The path-unawareness causes under-tainting. The key problem is that only one value for a constraint variable is recorded. The process of inferring critical bytes for the sample input is shown in the following steps: ① Record values: suppose that we record the latest value of each variable's occurrence, then after executing the sample input, the left-hand side values for $v1$ and $v2$ are "IDAT" and "IDAT". ② Perturb bytes and record new values: when the byte at 0x0c in Figure 1 is changed from "I" to "J", according to Listing 1, the path will change into "$v1 \rightarrow v2 \rightarrow v1$". The reason is that the modified chunk name "JHDR" is invalid, after parsing the valid chunk name "IDAT", the error in Line 9 ("Missing IHDR before IDAT") is triggered. After executing the mutated input, the left-hand side values for $v1$ and $v2$ become "IDAT" and "JHDR". ③ Compare values: because the value of $v1$ is unchanged while the value of $v2$ changes, the analysis will consider the byte at 0x0c as not critical to $v1$, but critical to $v2$. However, this byte is both critical to $v1$ and $v2$. More specifically, in the path "$v1 \rightarrow v2 \rightarrow v1 \rightarrow v2$", the byte is critical to the first occurrences of both $v1$ and $v2$, but not critical to their second occurrences. In more common cases, losing critical bytes would be more serious. More importantly, differentiating critical bytes for different occurrences of a single variable is important for fuzzers to explore more states.

Basic Idea of PATA. PATA performs path-aware taint analysis which locates critical bytes more accurately. The analysis process consists of the following steps: ① Collect the RVS of an execution path. PATA collects occurrences of each variable and records their values along the path into an RVS. Specifically, for the execution path [$v1$, $v2$, $v1$, $v2$], their respective left-hand side values are "IHDR","IHDR", "IDAT", and "IDAT". ② Perturb input bytes and record the RVSs under these inputs. As Figure 2 shows, when the byte at 0x0c in Figure 1 is changed from "I" to "J", the new occurrence sequence is [$v1$, $v2$, $v1$], and the left-hand side values are "JHDR", "JHDR", and "IDAT". ③ Match the variable occurrences between two RVSs. As shown in Figure 2, the first three occurrences of variables, namely $v1$, $v2$, and $v1$ are matched. The last occurrence of $v2$ in the original RVS does not appear in the new RVS. ④ Compare values between the matched occurrences. The first occurrences of both $v1$ and $v2$ change while the second occurrence of $v1$ does not change. As a result, PATA considers the byte at 0x0c as critical to the first occurrences of both $v1$ and $v2$. This byte does not affect the second occurrence of $v1$ because the value "IDAT" is not changed. Continuing the process, PATA can obtain the critical bytes for each occurrence of all appeared constraints in the sequence. The results are effective for fuzzing because the fuzzer can accurately mutate the critical bytes to explore any neighbor basic blocks it deems interesting along the path.

III. PATA DESIGN

Figure 3 presents the overall design of PATA, which contains three steps: 1) Constraint Variable Collection, 2) Critical Byte Identification, and 3) Path-Oriented Mutation. The following text in this section will present the details of each step.

A. Constraint Variable Collection

The first challenge in designing path-aware taint analysis is tracing *representative variables* reasonably. Formally, a *constraint variable* is defined as a tuple (V, P), where $V = \{V_1, ..., V_n\}$ is a set of program variables used in a constraint, and P is the predicate used in a constraint. For example, the constraint `a > 5` contains one constraint variable, whose $V = \{a, 5\}$ and $P = ' >'$ (*greater than*). To address this challenge, as shown in Figure 3, PATA identifies the representative constraint variables and records their values for each occurrence into the Representative Variable Sequence (RVS), which consists of all occurrences of representative variables visited in the execution path. To collect representative variables, PATA first collects all program variables used in constraints; for each variable found, PATA increases input mutation sensitivity by backtracking to its source variables that are directly influenced by input bytes if needed. Apart from identification, PATA extracts the variable features to provide more information for input mutation.

1) Identify Representative Variables: Constraint variables consist of variables that can influence a constraint directly or indirectly. To allow for path-awareness, the ideal strategy is to record all occurrences of encountered constraints variables and their values. But it is impractical and unnecessary to record all of them. To reduce the scope for observing value changes, PATA searches for all constraints in the programs to extract the variables which influence them directly. However, to infer critical bytes through mutating and monitoring values, the variables' value should be sensitive to inputs. In other words, when inputs have been mutated, their value should be changed. We denote these variables as *representative variables*. Constraints that take the form of logical operations and byte array comparisons require special attention.

Logical Operations: Constraints can take the form of logical operations. In other words, a constraint can be composed of the conjunction or disjunction of several basic comparisons. Because the constraint is dependent on other variables, its value is difficult to change. Even if we observe a change in the variable's value, it is difficult for us to mutate the affecting bytes because the constraint variable supplies little features. For these constraints, we should backtrack their sources to identify their dependent variables and obtain rich constraint features to assist mutation.

For example, the code snippet in Listing 2 is extracted from the function `png_read_info` in `libpng`. The constraint at Line 2 checks whether the color is a certain type and verifies whether the chunk `PLTE` is missing. This constraint is a logical operation, where its operand is a boolean variable that has only two possible values. If we directly monitor this variable, it will be difficult to observe value changes, preventing us from effectively locating critical bytes. The solution is to backtrack its source variables. The variable has two source variables, the first is related to the color type and the second is related to the state of the chunk parser. Both of the variables are more directly influenced by the input bytes. For example, when we take the sample file in Figure 1 as an input, the byte 0x19 in `IHDR` chunk determines the color type. By mutating each byte in the input, we can easily find any value changes and locate corresponding critical bytes. If source variables are still logical operations, we need to backtrack their sources recursively.

```
1  // ... png_read_info ...
2  else if (png_ptr->color_type == PNG_COLOR_TYPE_PALETTE
3         && (png_ptr->mode & PNG_HAVE_PLTE) == 0)
4     chunk_error(png_ptr, "Missing␣PLTE␣before␣IDAT");
5  // ...
```

Listing 2. An example for variable identification. The constraint is a logical operation whose value is difficult to change and has fewer features.

Byte Array Comparisons: Programs often use functions to compare the content of two byte arrays. Similar to logical operations, its comparison result is difficult to change. Even if the value changes under input perturbation, directly recording its result loses the contents which are essential for mutation. To overcome such constructs, we should check whether a variable comes from a byte array comparison and record its compared contents. To support both logical operations and byte array comparisons, we employ Algorithm 1 to identify source constraint variables.

Algorithm 1 takes the source code of a program as the input and outputs a vector of unique representative variables. It first scans all constraints and transforms them into constraint variables that influence the constraints directly. These variables are added to V without duplication. Next, each recorded constraint variable is checked: if it is a logic operation, we substitute it with all of its operand sources. If the constraint uses the results of byte-array compare functions, we substitute

Algorithm 1: Representative Variable Identification

```
   Input  : Source code: $src$
   Output : Vector of unique representative variables: $V$
1  $C$ = scanConstraint($src$);
2  for $c$ in $C$ do
3      recordAndTrans($c$,$V$);
4  end
5  while isChanged($V$) do
6      for $v$ in $V$ do
7          if isLogicOperation($v$) then
8              substitute($v$, operands($v$));
9          else if isByteArrayFunction($v$) then
10             substituteAsByteCmp($v$,
                parameter($v$));
11         end
12     end
13 end
```

it as a byte-array comparison variable to record its parameters. Algorithm 1 continues this process until there are no further changes for V.

2) Extract Features: As constraint variables are identified, their detailed *constraint variable features* are also extracted. Variable features consist of operand features (operand data type and length), pattern features (constraint category, e.g. `cmp`), and block features (dependent basic blocks of a constraint). These features supply more information for path-oriented mutation.

Operand features describe the operand data type and its bit length. The data type dictates how compilers or interpreters should handle its corresponding variable. We use three types to cover all possible cases in real-world programs, namely integer types, floating-point types, and byte array types. This allows the fuzzer to deduce precise runtime values when mutating seeds. In contrast, many other works ignore the data type and always interpret the operands as integers, which introduces inaccuracies during seed mutation. Apart from the data type, operand bit length is another important feature. The bit length reduces the solution space for fuzzers when solving some complicated constraints. It is also useful for fuzzers to determine the relationship between the critical bytes and the constraint variables like *direct copy*, i.e. the constraint value and critical bytes have a one-to-one mapping relationship.

Pattern features describe a constraint variable's category and its specific features. A constraint variable possesses one of the following three patterns, namely `cmp`, `switch`, and `call`. ① `cmp`: This is the most common pattern, which generally has two operands (lhs and rhs) in comparison. Its specific features consist of two elements: the predicate and whether it has a constant operand. The predicate describes how operands are compared. Recording whether a `cmp` has a constant value allows the fuzzer to bypass magic-byte-comparison constraints efficiently. If one of the operands is a constant, we always set it as the rhs. ② `switch`: This pattern matches one expression with several certain cases. If one of the cases is matched, the control flow is transferred to the corresponding destination of the case; otherwise, the control flow is transferred to the default destination. Each case in a `switch` comparison is always a constant, thus we directly record its data type and all its case values as its specific features. ③ `call`: This pattern represents the program call functions to compare two strings or the content of byte arrays. For this category, we record the actual function name as its specific feature.

Block features record information of the basic block that a constraint variable belongs to. Given the block features, we can find the successor blocks of the constraint and determine its mutation priority with its success blocks' coverage. Block features should be extracted along with the variable identification to maintain the corresponding relationship.

B. Critical Byte Identification

The second challenge is to match variable occurrences correctly when execution paths alter during input perturbation. To address this challenge, PATA locates critical bytes by byte-level mutation and occurrence matching, as shown in Figure 3. In byte-level mutation, PATA collects the RVSs consisting of variable occurrences across runs. PATA matches the RVSs of perturbed inputs with that of the original input to identify corresponding critical bytes for each variable occurrence.

Critical bytes represent the bytes that influence the value in a given occurrence of the constraint variable. PATA slightly mutates each byte of the input to obtain a new RVS and monitors the value changes. These mutations may result in the paths deviating from the original path, with the same constraint variable having different visit counts or being completely skipped. Directly abandoning these bytes will result in precision loss. This leads to a dilemma in inferring critical bytes. One constraint variable might appear several times in the sequence. If we only record the value of one occurrence of the constraint variable, we will lose value changes for other occurrences. But if we monitor each occurrence of one constraint variable, then when the execution path changes, one variable might appear in different places in the two sequences. To further complicate matters, single-byte mutation sometimes causes the number of occurrences to increase, decrease, or reduce to zero. Constraint variable mismatches result in the incorrect inference of critical bytes.

We take the motivating example in Section II which is extracted from `libpng` to demonstrate such issues in path matching. For each chunk in a `PNG` file, the code snippet compares its name with predefined values sequentially until the matched type is found and handles them with their corresponding logic. Suppose we use the sample input in Figure 1 and perturb the byte at offset 0x0c from ‘I’ to ‘J’. Figure 2 presents the execution path before and after the perturbation. The path changes because `libpng` does not allow an `IDAT` chunk to come before an `IHDR` chunk and the modified input triggers the corresponding error checking logic when dealing with the `IDAT` chunk. Although the path changes, we should match the common occurrence of variables, e.g. the first occurrence of $v1$ in both sequences.

We propose Algorithm 2 to address this problem. For each offset in the input, we modify them using each perturbation method, including bit flipping, incrementing or decrementing by one, and replacing it with some interesting values. The intention is to modify the byte by a small amount to maintain the execution path as much as possible but alter the value of byte dependent variables. However, some variables may be unstable, i.e. they may take different values in different runs with the same input. The algorithm will skip over

Algorithm 2: Critical Byte Identification

```
Input  : Input seed: $s$, Original path: $path$,
         Unstable variable identifier list: $unstable$
Output : Hashmap (occurrences→critical bytes): $C$
1  $vseq$ = getSeqForEachVar(RVS($path$));
2  foreach $offset \in s$ do
3      foreach $perturb\ method\ Opd$ do
4          $s'$ = perturb($s$, $offset$, $Opd$);
5          $path'$ = execute($s'$);
6          $vseq'$ = getSeqForEachVar(RVS($path'$));
7          foreach $v \in vseq.key$ do
8              if $v \in unstable$ then
9                  continue;
10             end
11             $occurSeq = vseq[v]$; $occurSeq' = vseq'[v]$;
12             $min\_len$ =
                 min(len($occurSeq$),len($occurSeq'$));
13             for $i = 0$ to $min\_len$ do
14                 if value($occurSeq[i]$) $\neq$
                     value($occurSeq'[i]$) then
15                     $occur = occurSeq[i]$;
16                     $C[occur] \cup = offset$;
17                 end
18             end
19         end
20     end
21 end
```

them to avoid over-tainting. By monitoring value changes, the dependent critical bytes are identified. However, the path may change under perturbation. To match variables effectively when the path changes, we introduce a data structure called the *Variable Occurrence Subsequence*. It is a map where the key is the variable identifier and the value is the occurrence sequence in the path. The algorithm considers the occurrences in the common prefix that are matched and compares their values. If the value changes between the matched variables, the changed byte is inferred as critical to the occurrence.

We demonstrate the aforementioned steps using the example in Section II. After perturbing the bytes with offset 0x0c, PATA extracts occurrences of each constraint variable in the RVS into a subsequence. For $v1$, the original left-hand side value of the subsequence is [“IHDR”, “IDAT”], and the new subsequence becomes [“JHDR”, “IDAT”]. Following the algorithm, as shown in Figure 2, the two occurrences of $v1$ are both matched between changed paths. By comparing the matched values, we infer that the byte at 0x0c is critical to the first occurrence of $v1$. This byte is not critical for the second occurrence because the value “IDAT” is not changed. Proceeding with the process, PATA can get the critical bytes for each occurrence of all visited constraint variables in the RVS. The results are useful for fuzzing because the fuzzer can accurately mutate critical bytes to explore other branches of each occurrence along the execution path.

C. Path-Oriented Mutation

The third challenge is to exploit analysis results effectively, which implicates path-awareness mutation. As shown in Figure 3, PATA employs a path-oriented mutation method to exploit the analyzed results. For each seed, we can obtain the RVS consisting of constraint variables and their respective values. At each entry along the sequence, our objective is to mutate certain bytes of the input to explore undiscovered program states. Using the critical bytes for each occurrence of constraint variables in a sequence, combined with occurrence values and detailed features, PATA is capable of performing a more precise mutation than random fuzzing.

We take the example in Section II to show how the mutation exploits path-awareness to great effect. The sample input has the occurrence sequence [$v1$, $v2$, $v1'$, $v2'$] and corresponding block sequence [$B4$, $B2$, $B5$, $B1$, $B3$]. Path-aware taint analysis discovers the critical bytes for four constraint variable occurrences as 0x0c-0x0f, 0x0c-0x0f, 0x25-0x28, and 0x25-0x28, respectively. Along the variable occurrences in the sequence, the path-oriented mutation performs the following steps: ① For $v1$, it changes the value of 0x0c-0x0f from "IHDR" to "IDAT" to explore the new path [$B5$, $B6$]; ② For $v2$, it changes the value of 0x0c-0x0f from "IHDR" to another value like "AAAA" to explore the new path [$B4$, $B3$, $B5$, $B6$]; ③ For $v1'$, it changes the value of 0x25-0x28 from "IDAT" to another value like "AAAA" to cover the new path [$B4$, $B2$, $B4$, $B3$]; ④ For $v2'$, it changes the value of 0x25-0x28 from "IDAT" to "IHDR" to execute the path [$B4$, $B2$, $B4$, $B2$]. This process relies on path-aware taint analysis. Because path-unaware taint analysis will regard $v1$ and $v1'$ as the same, then it will only infer the bytes 0x25-0x28 are critical for $v1$. By mutating them, only one new path could be obtained: [$B4$, $B2$, $B4$, $B3$]. Thus, we may lose bugs that are implicated in other paths.

Algorithm 3: Path-Oriented Mutation

```
Input : Input seed: $s$, Path: $path$,
        Critical byte hashmap: $C$,
        Constraint variable features: $B$,
        Coverage tracking program: $P$,
        Value tracking program: $P'$
for $v \in$ RVS($path$) do
    if isMeaningfulToMutate($v$) then
        if lengthExplore($s,v,B,P$) then
            continue;
        end
        $cbytes = C[v]$;
        if isEmpty($cbytes$) then
            continue;
        end
        if copyExplore($s,cbytes,v,B,P$) then
            continue;
        end
        if linearSearch($s,cbytes,v,B,P,P'$) then
            continue;
        end
        randomExplore($s,cbytes,v,B,P$);
    end
end
```

Algorithm 3 presents the overall process of path-oriented mutation. For each occurrence of constraint variables along the RVS, we first check whether it is meaningful to solve. For instance, if all the successor basic blocks of this variable have been covered or the value of the constraint variable is not stable, we consider it to be not meaningful. We aim to solve the unexplored branches of the occurrence. Then we explore whether the constraint variable is related to the input length. If so, we will try to increase or decrease input length to satisfy the constraint. Next, we check whether the critical bytes for the occurrence have been successfully located. If not, we skip and proceed with the next constraint. Using the critical bytes, values, and features of a given constraint variable, we mutate the input using the following methods in turn: direct copy exploration, linear search, and random exploration. The mutated seeds will be passed to a conventional fuzzer to execute with a coverage tracking binary and examined if they have triggered new coverage or exhibited abnormal behaviors. If so, they will be saved for further mutation or anomaly analysis. If any mutated seeds successfully cover the target branch, we then move on to the next occurrence in the RVS.

1) Length Exploration: The length of the input is an important argument for fuzzed programs. Many program states could only be triggered when the length meets some preset conditions. However, many conventional fuzzers prefer short inputs to boost execution speed. In addition, they do not have the means to identify the relationship between the input length and the constraint variables. Therefore, they can only rely on random methods to modify the input length.

PATA obtains detailed features of the constraints variables, which allows the fuzzer to explore input length related constraints. Specifically, length related constraint variables have a high probability of exhibiting the pattern `cmp` and `switch`. For a `cmp` constraint variable, if rhs (right-hand side) is a constant (call back functions always pass the constant value to rhs), we check whether the value of lhs (left-hand side) equals the input length. If so, based on the predicate, current length, and the constant target value, we could precisely append bytes or delete bytes to reach the expected length. For a `switch` variable, we check whether its runtime value equals the length. If so, we regard each value of its cases as the target value, calculate the expected length, and append or delete bytes.

2) Direct Copy Exploration: The constraint variable and its critical bytes might have many complex relationships, one of which is "direct copy", i.e. the input bytes are directly used in constraints variables. Magic numbers and magic bytes are commonly recognized roadblocks of fuzzing, and exploring the direct copy is a decisive factor to satisfy them.

Direct Copy Identification. The process of identifying consists of the following steps: ① transform the operands into a byte sequence, ② split the critical bytes into several continuous sections, and ③ find the byte sequence in these sections. In the first step, the main issue is determining endianness, i.e. the ordering of bytes in a multi-byte numerical type. Intuitively, we should use the endianness of the underlying system. However, we find that many projects differ from the system endianness. Thus, we transform the operands into both little-endian and big-endian byte sequences for the following process. If one of the sequences matches the input bytes, we record the matched endianness. For simplicity, we also drain the extra zero bytes in its prefix in the big-endian sequence or suffix in the little-endian sequence.

The critical bytes reflect the influence of certain input bytes on the constraint variables. If the operand has a one-to-one correlation with certain input bytes, then these bytes are most likely copied in a continuous section from critical bytes. Consequently, in the second step, we split the dependent critical bytes into several sections based on continuity. In the final step, we search the little-endian or big-endian operand byte sequences in each section. If one such input section is found, then a direct copy relationship is confirmed. If it appears several times, then all occurrences would be recorded.

More specifically, the identifying process of different constraint variables exhibits some differences. For `cmp` pattern constraint variables, they generally have two operands (lhs and rhs) in comparison and a predicate that specifies the comparative operation. The constraint feature indicates whether the constraint has constant values and if so, the callback function will always put the constant to rhs. In this case, we directly check lhs. Otherwise, we should check both lhs and rhs. For `switch` pattern constraints, their operand runtime value controls the execution transforms to which case branch. Because the cases are always constant, we should only check whether its operand is a direct copy. There are some small differences in the `call` pattern constraints. Its operands are two byte-sequences, so we skip step 1. Because both operands might be a direct copy, we need to check both.

Expected Value Calculating. The identified direct copy implies the source of the operands. The next step is calculating the expected value to cover the target block.

For `cmp` patterns, the constraint feature indicates the predicate and the operand data type. We can calculate the expected value by combining the runtime values of lhs and rhs. There are mainly three cases. ① rhs is a constant and lhs is a direct copy of certain input bytes. In this case, we should set the lhs based on the predicate and operand data type. Using the code in Listing 3 as an example, assume the runtime value of lhs and rhs is 13 and 15. From the predicate "less than" of constraint feature, we get that the condition is satisfied by the current value because the lhs is less than rhs. Therefore, we need to calculate the lhs value which is equal to or greater than rhs. To calculate a greater value, we simply add 1 to the constant value based on the data type. Consequently, the target value of lhs is 15 or 16. ② rhs is not a constant and both lhs and rhs are direct copies of the input. In this case, we would regard the rhs as the constant, maintain the value of the rhs, and only change lhs as we did in the previous case. ③ rhs is not a constant and only one of the two operands is the direct copy of the input. In this case, we would regard the operand which is not copied from the input as a constant and change the other one as performed in the first case.

```c
void Fuzz(char * input, int len) {
    if(*(uint64_t *)(input) < 15) { /*...*/}
}
```

Listing 3. The left-hand side directly uses the first 8 bytes of the input.

For `switch` patterns, the constraint features record values of each case. Each case is a constant, and its predicate could always be regarded as "equal", so the expected value is just the value of each case.

For `call` patterns, the constraint features indicate the name of the called buffer comparison functions. We divide these functions into two categories: ① comparing whether the contents of the two buffers are the same, like "bcmp", or ② checking whether the contents of one buffer contains another, like "strstr". No matter in which category, when two buffers' contents differ, we could use the contents of one buffer which is not copied from the input as the expected value.

Critical Byte Patching. After calculating the expected value, we patch the critical bytes to reach the value. Based on the endianness previously identified, we transform the expected value into byte sequences and replace the corresponding bytes in the input with the required values. Sometimes the length of byte sequences may be longer than the direct copy bytes, but we still try to replace them for fault-tolerance.

3) Linear Search: Apart from the direct copy relationship, other input-to-variable relationships are more complex. Among them, one common and solvable relationship is the monotonic relationship between the critical bytes and the constraint variable. In this mutation method, we view the constraint as a function, whose argument is the vector of critical bytes and the value is the gap. The gap measures the difference between the current state and the target state that satisfies the constraint. For these constraints which could be seen as monotonic functions, linear search is a practical way to solve them. Although some functions are not monotonic globally, they may exhibit local monotonic behavior. Linear search also has the possibility to solve them under these situations. Even if they can not be directly solved, linear search is also helpful to find a temporary seed that is closer to the target value. Based on the temporary seed, PATA is more likely to cover the targets. Because buffer comparisons in `call` pattern constraint variables generally are hard to view as monotonic functions, we only employ linear search on `cmp` and `switch` pattern constraints.

Algorithm 4: Linear Search

```
Input : Input seed: $s$, Critical bytes: $cbytes$,
        Variable runtime value: $v$,
        Constraint variable features: $B$,
        Coverage tracking program: $P$,
        Value tracking program: $P'$
1  $gap$ = getGap($v$, $B$); $ops$ = []; $differences$ = [];
2  foreach $b \in cbytes$ do
3      calMoveOperation($b$,$gap$,$ops$,$differences$,$P'$);
4  end
5  $scbytes$ = sortByDifference($cbytes$,$differences$);
6  foreach $b \in scbytes$ do
7      $op = ops[b]$;
8      if isStandStill($op$) then
9          break;
10     end
11     while True do
12         $s$ = move($s$,$b$,$op$);
13         $v'$ = executeGetValue($s$,$P'$);
14         $gap'$ = getGap($v'$,$B$);
15         if hasSolved($gap'$) then
16             executeCheckCoverage($s$,$P$);
17             return;
18         end
19         if $gap' \geqslant gap$ then
20             break;
21         end
22         $gap = gap'$;
23     end
24     restoreLast($s$);
25 end
26 executeCheckCoverage($s$,$P$);
```

Algorithm 4 illustrates the process of linear search. We treat each critical byte as an independent variable and linearly increment or decrement the value of bytes in turn to find a suitable answer. The process consists of the following steps:

i) Calculate the gap. The gap measures the difference amount between the current value and the target state that will solve the constraint. For `cmp` patterns, the gap is calculated by obtaining the absolute of the difference between lhs and rhs. For `switch` patterns, we regard it as multiple constant `cmp` patterns. The gap is calculated by obtaining the absolute difference between the control value and the case values. The constraint variable feature indicates the data type at runtime, so we can get the precise type-aware gap value.

ii) Determine the movement operations. Each byte may have three movement operations: increase, decrease, or standstill. The next task is to determine the movement operations for each byte. First, we add or subtract a small value (e.g. 1) to the byte and calculate new gaps. Second, we compare the gaps to determine the operation. The basic rule is choosing the move direction whose gap is smaller both than the other and the original gap. If neither of them has a smaller gap, we set the operation as standstill. Sometimes the gap might be invalid, for example, the variable disappears because of the path change caused by an earlier conditional statement. When the invalid gap is compared, we always regard it as bigger than others. We call the changes for the gap before or after the operation as the difference. The mutation difference applied for the standstill operation is always 0.

iii) Sort critical bytes by differences. The bytes whose differences are greater will move first.

iv) Move and Search. For each byte in the sorted vector, we move it according to its required operation, acquire the new value, and calculate the new gap until the gap does not decrease. If any of the mutated seeds solves the constraint, we then conclude the process. If the operation of any bytes is "standstill", all succeeding bytes are also regarded as "standstill" because they are sorted, allowing us to short-circuit to the end of the loop. When the constraint is not solved after the whole process, we still execute and save the final mutated input for further mutations.

4) Random Exploration: If all of the previous methods do not satisfy the constraint, we still try our best to randomly mutate the critical bytes. The mutation starts with the final mutated input from linear search. Specifically, we first split the critical bytes into several sections according to continuity. For each section, we randomly choose several bytes to mutate, operations to mutate, and the number of rounds to mutate. The operations include bit flipping, interesting values replacing, and arithmetic operation. The mutations might still not pass the constraint, but its side effect may help PATA find more uncovered basic blocks.

IV. Implementation

As Figure 3 shows, the constraint variable collector, critical byte locator, and the path-oriented mutator are the three main components of PATA. The constraint variable collector is implemented using LLVM and Clang with over 2000 lines of C++ code. The critical byte locator and the path-oriented mutator are implemented along with other necessary fuzzer components in over 20000 lines of Rust code. Details are illustrated as follows.

The constraint variable collector is performed based on LLVM and Clang. PATA modifies the compiler driver (i.e. clang and clang++) to output LLVM IR (Intermediate Representation) and combine all IR files into one module. Then the module is analyzed to identify basic blocks and produce a unified intermediate representation. Then, PATA employs Algorithm 1 to identify variables. Specifically, PATA scans for constraints by searching and backtracking all "icmp" or "switch" instructions. In addition, we summarize 9 byte-array compassion functions, namely "bcmp", "memcmp", "memmem", "strncmp", "strncasecmp", "strcmp", "strcasecmp", "strstr", and "strcasestr". For these functions, PATA backtracks and records their parameters as a constraint variable which has the `call` pattern. The block features of the constraints are also extracted at the same time.

The critical byte locator implements Algorithm 2. For a new input seed, it perturbs each byte with bit flipping, interesting values replacing, and arithmetic operation. The path-oriented mutator implements Algorithm 3. For each input seed, it analyzes along its path and acquires constraint variables that are meaningful to solve. Then it mutates critical bytes of the constraint variables with the aforementioned methods.

The collector, locator, and mutator are supported by low-level components, mainly consisting of the communication and executor components. The communication component exchanges information with the target process by sockets and shared memory. The executor component provides low-level support for running one input. It implements the "forkserver" technique for efficiency. The variable value tracking executor stores the runtime value of each variable occurrence inside the shared memory, and the coverage tracking executor also places a counter for basic blocks in it. Apart from the taint analysis driving algorithm, we also implement necessary components like the selector and the saver component to manage input seeds for conventional fuzzing. They follow the algorithm of AFL: the selector prefers short and fast seeds, and the saver promotes an input into the corpus if new coverage is observed.

V. Evaluation

We first evaluated PATA on two widely used benchmarks: Google's fuzzer-test-suite which contains a series of real-world programs and LAVA-M which consists of four programs taken from `coreutils` inserted with synthetic bugs. Then we used PATA to test more popular open-source projects obtained from GitHub. PATA found 40 previously unknown bugs. Furthermore, we also evaluated the effectiveness of path-awareness and its overhead to comprehensively understand the improvements of PATA.

A. Evaluation Setup

Experiment environment. All experiments were conducted on a machine running 64-bit Ubuntu 18.04 with 80 cores (Intel(R) Xeon(R) Gold 6148 CPU @ 2.40 GHz) and 320 GiB of main memory. We ran each fuzzer on each target application in identical configurations, specifically single-threaded execution on one CPU core over a period of 24 hours. All experiments were repeated 5 times.

Compared fuzzers. In our evaluation, PATA was compared to AFL [36], MOPT [24], TortoiseFuzz [33], VUzzer [26], Angora [11], REDQUEEN [6], and GREYONE [17]. AFL is a classic mutation-based greybox fuzzer that many others have extended upon. MOPT and TortiseFuzz are the two most recent proposed works which are based on AFL. VUzzer, Angora, REDQUEEN, and GREYONE are compared because they are the most related works that combine taint analysis with fuzzing. Since REDQUEEN is targeted at OS kernels, we use the REDQUEEN mode of AFL++ [16] for fairness. As the source code of GREYONE is not available, we reimplement its taint inference, byte prioritization, and conformance-guided evolution on PATA's framework.

Initial input seeds. For Google's fuzzer-test-suite and LAVA-M, all fuzzers used the same seeds collected from the data set. When there was no seed available, an empty seed was used as a fallback option. For real-world projects, we randomly selected one valid input obtained from the Internet.

Performance metrics. We evaluated fuzzers using three metrics, namely the number of paths executed, basic blocks covered, and bugs triggered. Because different fuzzers may have different representations of fuzzing states, in order to have a fair comparison, we collected and ran their seeds generated over the course of evaluation to gather the number of paths (identified by AFL's algorithm) and basic blocks (identified by LLVM tools) to unify these representations.

Another aspect is the number of bugs triggered. The method of distinguishing unique crashes also varies between different fuzzers. To have a fair and better comparison, we use the number of bugs identified from these crashes as a metric. For LAVA-M, we identify unique bugs by the printed ID when the bug is triggered. For others, we define a bug as **unique** when it has a different call stack when compared to others. We distinguish bugs that come from crashes in two steps. First, we collect the triggered crash inputs, re-execute and filter them by backtracking the call stack. To improve accuracy, we further analyze the bugs manually to eliminate duplicate entries.

B. Evaluation on Google's fuzzer-test-suite

To demonstrate PATA's practical performance, we employ the real-world project benchmark, Google's fuzzer-test-suite, for evaluation. The binaries for all fuzzers except VUzzer are built with AddressSanitizer (ASAN) [27] enabled. VUzzer uses non-sanitized binaries because its taint analysis does not work on binaries compiled with ASAN.

Coverage. Tables I and II show the number of paths and blocks covered by each fuzzer. PATA found 111%, 63%, 125%, 1830%, 343%, 43%, and 29% more paths, covered 19%, 12%, 21%, 87%, 41%, 7%, and 19% more basic blocks compared to AFL, MOPT, TortoriseFuzz, VUzzer, Angora, REDQUEEN, and GREYONE, respectively. These statistics illustrate that PATA improves performance in most of the projects on two metrics, especially in paths. To ensure the findings are not the result of statistical errors, Tables XIII and XIV also show the p-value between PATA and other fuzzers regarding the total number of paths and basic blocks for all projects in 5 repetitions. They show that for most cases, the differences are significant ($p < 0.05$). To show their performance over an extended period, we also extend the experiment to 60 hours. Table X and Table XI in Appendix A-F show that PATA also outperforms other fuzzers. Specifically, PATA found 101%, 68%, 132%, 2209%, 390%, 66%, and 49% more paths, covered 21%, 17%, 29%, 101%, 49%, 14%, and 20% more basic blocks than AFL, MOPT, TortoriseFuzz, VUzzer, Angora, REDQUEEN, and GREYONE, respectively.

TABLE I
AVERAGE NUMBER OF PATHS OVER 5 RUNS IN 24 HOURS

Project	AFL	MOPT	TortoiseFuzz	VUzzer	Angora	REDQUEEN	GREYONE	PATA
boringssl	380	486	395	164	270	471	486	662
c-ares	25	25	26	27	19	29	35	34
freetype2	2868	6348	3246	60	2868	6452	9460	9045
guetzli	889	1181	763	31	918	587	1317	2006
harfbuzz	3215	4248	2371	137	1729	4415	4488	5578
json	476	519	443	23	260	604	865	995
lcms	210	208	213	12	340	449	179	775
libarchive	1287	2106	1247	79	1089	2073	1598	3752
libjpeg	990	1491	987	26	741	1349	1440	2643
libpng	298	330	323	17	305	377	509	628
libssh	18	18	18	14	14	366	25	484
libxml2	1530	2350	1753	194	725	2705	2409	6101
llvm-libcxxabi	2245	3154	2444	189	658	4818	6128	6753
openssl-1.0.1f	271	303	218	12	39	240	199	408
openssl-1.0.2d	693	623	667	363	366	731	819	851
openssl-bignum	328	346	324	417	370	332	384	388
openssl-x509	876	881	862	843	858	864	939	966
openthread	586	554	585	35	329	859	818	766
pcre2	11598	11586	9794	325	1748	15445	15529	19318
proj4	70	72	66	20	106	204	108	720
re2	1920	1970	1833	285	1012	1964	2827	2811
sqlite	234	246	230	82	172	248	267	300
vorbis	639	776	662	51	578	690	974	1270
woff2	512	517	555	62	7	456	833	827
wpantund	847	2392	959	143	205	1852	1485	1610
Total	33005	42730	30984	3611	15726	48580	54121	69691
Improve	111%↑	63%↑	125%↑	1830%↑	343%↑	43%↑	29%↑	–
p-value	0.006	0.006	0.006	0.006	0.006	0.006	0.006	–

TABLE II
AVERAGE NUMBER OF BASIC BLOCKS OVER 5 RUNS IN 24 HOURS

Project	AFL	MOPT	TortoiseFuzz	VUzzer	Angora	REDQUEEN	GREYONE	PATA
boringssl	2216	2254	2203	2191	2211	2238	2204	2269
c-ares	63	63	64	64	63	64	66	66
freetype2	10700	12719	10770	5908	10258	14372	13311	15325
guetzli	6193	6283	6107	5408	6188	6034	6178	6290
harfbuzz	9373	9827	8702	5763	8280	9665	9546	9924
json	1249	1264	1196	413	1199	1587	1580	1607
lcms	1661	1662	1656	658	2047	2251	1227	2501
libarchive	3494	4320	3222	1191	4460	4181	3195	6295
libjpeg	2367	2718	2339	1213	2321	2511	2393	2822
libpng	1171	1182	1154	677	1336	1392	1194	1413
libssh	898	898	898	898	852	1618	908	1611
libxml2	4342	5140	4625	2587	3650	4958	4356	8243
llvm-libcxxabi	3575	3657	3566	1812	2818	3850	3742	3952
openssl-1.0.1f	4937	4689	4743	896	1974	5098	2220	5094
openssl-1.0.2d	1408	1411	1396	1343	1092	1412	1386	1394
openssl-bignum	1100	1105	1102	1110	1107	1100	1111	1113
openssl-x509	5304	5304	5281	5281	5312	5305	5347	5349
openthread	5562	5592	5536	3238	5124	6798	5628	5619
pcre2	8830	8673	8124	2268	4407	9293	9022	9494
proj4	202	202	209	141	572	717	209	1752
re2	5587	5582	5575	4092	4797	5584	5654	5663
sqlite	1565	1805	1620	1404	1511	1805	1804	1808
vorbis	2030	2056	2001	1292	1885	1957	1942	2071
woff2	2990	2994	2997	2683	632	2991	3058	3061
wpantund	11795	13409	12105	10403	9458	13166	11473	13019
Total	98612	104809	97191	62934	83554	109947	98754	117755
Improve	19%↑	12%↑	21%↑	87%↑	41%↑	7%↑	19%↑	–
p-value	0.006	0.006	0.006	0.006	0.006	0.006	0.006	–

AFL, MOPT, and TortoiseFuzz obtain little information about the target program. They mutate seeds randomly and evolve the seed queue guided by coverage. In contrast, PATA not only recognizes the critical bytes for each occurrence of constraint variables but also obtains their runtime value. Based on the dependent critical bytes, dynamic runtime values, and static variable features, PATA mutates seeds to explore each uncovered branch in the path effectively. It is noteworthy that PATA has several projects (e.g. `opensssl-1.0.2d`, `wpantund`) whose number of paths found or basic blocks covered are close to those of AFL and MOPT. This may originate from unstable paths in these projects. In most cases, sending the same input multiple times should take the exact same path every time. However, the behavior of programs can be influenced by randomness, e.g. reaction to timing, etc. Thus, in some of the re-executions with the same input, the path will be different across runs. PATA skips the input seeds when their paths are unstable. Consequently, PATA resorts to a normal fuzzer, and it performs close to AFL and MOPT.

Compared to the other four taint analysis assisted fuzzers VUzzer, Angora, REDQUEEN, and GREYONE, PATA also has better coverage. The main reason is PATA identifies dependent critical bytes more accurately with path-aware taint analysis. VUzzer and Angora may over-taint or under-taint due to their inaccurate taint analysis approaches. REDQUEEN only locates direct copies with costly colorization. GREYONE might less locate or mismatch critical bytes because of multiple occurrences. In contrast, PATA employs a path-aware taint analysis. It locates critical bytes based on the value change of matched occurrences of constraint variables under byte-level mutation, which overcomes the obstacles for taint propagation. More importantly, PATA locates dependent critical bytes for each variable occurrence in the path, avoiding issues caused by loops or multiple calls. In addition, PATA mutates seeds with detailed data flow features in multiple ways. For constraints that are related to the input length or direct copies, PATA solves them effectively. For other more complicated constraints, PATA also searches answers. With the data types in constraint features, PATA could acquire precise feedback, which helps it find solutions more effectively.

Bug Triggering. The coverage improvement increases PATA's possibility of finding bugs. Table III presents the number of bugs triggered by each fuzzer. The bugs are distinguished by comparing the call stack and conducting manual analysis. According to our statistics, VUzzer could not find any bugs, Angora only found 2 bugs on 2 projects, while AFL, MOPT, TortoiseFuzz, REDQUEEN, and GREYONE triggered 16, 21, 18, 18, and 9 bugs, respectively. PATA found 31 bugs on 11 projects. Google's fuzzer-test-suite lists some bugs which could be found for its projects. It is worth noting that PATA not only found the bugs listed in fuzzer-test-suite, but also found 1, 1, 5, and 10 unlisted bugs in project `json`, `libxml2`, `llvm-libcxxabi`, and `pcre2`, respectively. Listing 5 in Appendix A-F illustrates the details of one bug in `libxml2` which is only found by PATA. Table XII also shows that, in the 60-hour-long experiment, PATA triggers 14, 9, 13, 33, 31, 16, and 24 more bugs than AFL, MOPT, TortoiseFuzz, VUzzer, Angora, REDQUEEN, and GREYONE, respectively.

TABLE III
NUMBER OF BUGS OVER 5 RUNS IN 24 HOURS

Project	AFL	MOPT	TortoiseFuzz	VUzzer	Angora	REDQUEEN	GREYONE	PATA
c-ares	1	1	1	0	1	1	1	1
guetzli	1	1	1	0	0	0	0	1
json	1	1	1	0	0	1	1	2
lcms	1	1	1	0	0	0	0	1
libxml2	1	2	1	0	0	1	0	3
llvm-libcxxabi	4	7	6	0	0	7	1	7
openssl-1.0.1f	1	1	1	0	1	1	0	1
openssl-1.0.2d	1	1	1	0	0	1	0	1
pcre2	4	5	4	0	0	5	4	12
re2	0	0	0	0	0	0	1	1
woff2	1	1	1	0	0	1	1	1
Total	16	21	18	0	2	18	9	31
Increase	15↑	10↑	13↑	31↑	29↑	13↑	22↑	–

The bugs in Google's fuzzer-test-suite are complicated and do not have many common features. VUzzer did not find any bugs for two reasons. First, it does not support ASAN, so it could not detect any bugs outside of program crashes. Second, the relatively low coverage caused by the inaccurate taint analysis limits its ability to trigger bugs. In real-world projects which do not have as many magic bytes or values, random mutation also works well. This randomness allows AFL, MOPT, and TortoiseFuzz to trigger crashes with higher probabilities. Different from the AFL-family fuzzers, Angora only uses gradient descent on critical bytes to mutate inputs. Its mutation method is more deterministic, which restricts its ability to trigger crashes. REDQUEEN is able to solve direct-copy-related constraints. But it experiences difficulties in finding bugs behind constraints of other types. GREYONE cannot distinguish between variable occurrences, thus it may locate inaccurate critical bytes.

In contrast, PATA is more effective in triggering bugs. First, PATA utilizes path-aware taint analysis to locate more accurate critical bytes for each occurrence of a given constraint variable. Second, its path-oriented mutation tries to maximize the exploitation of one input by exploring all neighboring branches along the input's path. Covering more program logic correlates to having more chances of triggering crashes and finding bugs. Furthermore, PATA can still resort to random mutation, which increases the randomness and the possibility of triggering crashes. From the statistics, we conclude that PATA is not only able to improve coverage statistics, but also improves the possibility of triggering bugs.

C. Evaluation on LAVA-M

LAVA-M is an artificially constructed benchmark intended for evaluating the effectiveness of fuzzers, which has been used to evaluate many fuzzers such as VUzzer and Angora. It injects a large number of realistic bugs into four GNU `coreutils` programs: `base64`, `md5sum`, `uniq`, and `who`. Each injected bug is assigned a unique ID and the ID is printed when the bug is triggered. Evaluating on LAVA-M demonstrates the ability of fuzzers to explore program state space and trigger bugs.

Tables VII and VIII in Appendix A-D present the number of paths and basic blocks covered by fuzzing four projects in LAVA-M. They show that PATA performs the best in all four projects. On the LAVA-M data set, PATA executed 149%, 126%, 156%, 446%, 146%, 32%, and 6% more paths, found 68%, 68%, 68%, 67%, 12%, 7%, and 56% more basic blocks than AFL, MOPT, TortoiseFuzz, VUzzer, Angora, REDQUEEN, and GREYONE, respectively.

Table IV shows the range for the number of bugs detected by each fuzzer in 5 runs. The results show that PATA is more efficient in finding bugs in LAVA-M than other fuzzers. Specifically, PATA found 2593, 2564, 2593, 2474, 361, 2, and 1 more bugs than AFL, MOPT, TortoiseFuzz, VUzzer, Angora, REDQUEEN, and GREYONE, respectively. Additionally, PATA found these bugs very quickly. For `base64`, `md5sum`, and `uniq`, PATA found all listed bugs in less than 5 minutes.

Different from real-world programs, the bugs artificially injected in LAVA-M are triggered when their guarded magic value constraints are solved. Most of these constraints only occur once. In such a situation, random mutations of AFL, MOPT, and TortoiseFuzz cannot generate magic values and trigger bugs effectively. VUzzer utilizes taint analysis to find more bugs than AFL, MOPT, and TortoiseFuzz. However, its instruction-level dynamic taint analysis might be rather slow and inaccurate. Angora, REDQUEEN, and GREYONE perform better than VUzzer to find bugs in LAVA-M. In particular, REDQUEEN and GREYONE found almost all of the bugs in LAVA-M. This is because they can find more accurate critical bytes. These constraints are almost all related to magic bytes which come from direct copies. However, they will still suffer from over- or under-tainting for other constraints in more

TABLE IV
NUMBER OF BUGS (ACCUMULATED AND RANGE) DETECTED BY VARIOUS FUZZERS IN 5 RUNS WITHIN 24 HOURS

Project	Listed bugs	AFL	MOPT	TortoiseFuzz	VUzzer	Angora	REDQUEEN	GREYONE	PATA
base64	44	5 [3,5]	25 [10, 25]	4 [3, 4]	29 [19, 20]	46 [44, 46]	48 [44, 48]	48 [45, 48]	48 [48, 48]
md5sum	57	1 [0,1]	1 [0, 1]	0 [0, 0]	42 [18, 31]	54 [52, 54]	61 [54, 61]	61 [51, 61]	61 [57, 61]
uniq	28	0 [0,0]	3 [1, 3]	2 [0, 2]	27 [25, 26]	29 [28, 29]	29 [23, 29]	29 [26, 29]	29 [29, 29]
who	2136	3 [1,3]	9 [6, 9]	3 [2, 3]	30 [10, 13]	2112 [808, 2112]	2462 [1957, 2462]	2463 [2250, 2463]	2464 [2273, 2464]
Total	2265	9	38	9	128	2241	2600	2601	2602
Increase	–	2593↑	2564↑	2593 ↑	2474↑	361↑	2↑	1↑	–

complex projects (e.g. programs from Google's fuzzer-test-suite) because of path-unawareness.

PATA shows better performance than other fuzzers. With path-aware taint analysis, it identifies critical bytes more accurately. Based on path-oriented mutation, PATA is quicker to cover more states, solve the guard constraints, and trigger more bugs in LAVA-M. However, it is worth mentioning that the evaluation of fuzzers on LAVA-M might be less close to reality than on Google's fuzzer-test-suite, because all its bugs are inserted artificially, and each original project in LAVA-M contains less than one thousand lines of code.

D. Fuzzing Real-world Programs

We use PATA to fuzz more projects obtained from GitHub. Our statistics show that PATA delivers excellent performance. In total, it found 40 unknown bugs, in which 12 bugs have been assigned CVEs, as shown in Table V. We also used other fuzzers to test these projects but they only find a proportion of these bugs, as shown in Table IX in Appendix A-E.

TABLE V
THE BUGS DETECTED BY PATA

Project	Bugs	CVEs	Bug Type
bigint	1	-	segmentation fault
bitmap	1	1	segmentation fault
cyclonedds	9	-	buffer overflow, segmentation fault
ffjpeg	1	1	floating point exception
genann	2	2	buffer overflow, segmentation fault
jpeg_encoder	3	2	buffer overflow, segmentation fault
jpeg-compressor	6	2	buffer overflow
json	1	1	buffer overflow
libconfig	3	-	memory leak
libucl	1	-	assertion failure
libpng	3	2	memory leak, segmentation fault
lightmatrix	3	-	buffer overflow, segmentation fault
mxml	3	-	buffer overflow, segmentation fault
pdfalto	1	1	buffer overflow
SmallerC	1	-	buffer overflow
xml2json	1	-	memory leak
Total	40	12	-

Some of these bugs are hard to find. Let us use a memory leak bug found in png2pnm of libpng as an example, as shown in Listing 4. The bug is triggered when the constraint in Line 10 is passed through but the previously requested rpointers (Line 5) memory is not released. The constraint variable in Line 10 occurs multiple times, which is difficult for taint-based fuzzers to find the precise bytes in inputs in practice. Although solving the constraint seems to be trivial, with more or fewer bytes marked as critical, hardly can path-unaware fuzzers mutate the bytes efficiently and find this bug.

We also use AFL, MOPT, TortoiseFuzz, VUzzer, Angora, REDQUEEN, and GREYONE to fuzz this program, but they could not trigger this bug. Figure 4 illustrates the growing trends of the number of paths executed and basic blocks covered over 5 runs of 24-hour experiments. Based on path-aware

```
1  if (setjmp (png_jmpbuf(png_ptr))) {
2    png_destroy_read_struct(/*...*/);
3    return FALSE; // <= Memory leak happens
4  } // ...
5  if ((rpointers = (png_byte **)
6    malloc (/*...*/) == NULL) {// Allocate memory ...
7  } // ...
8  for (j = 0; j < pass; j++) {
9    for (i = 0; i < image_height; i++) {//...
10     if (length > buf_state->bytes_left) {
11       //Jump to Line 3 but rpointers is not released
12       png_error(png_ptr, "read_error");
13     } //...
14   }
15 }
```

Listing 4. A memory leak detected by PATA.

taint inference, PATA performed better than other fuzzers and remained ahead most of the time on both metrics. In addition, the shaded areas of PATA are small, which represents that PATA performs more stable than other fuzzers.

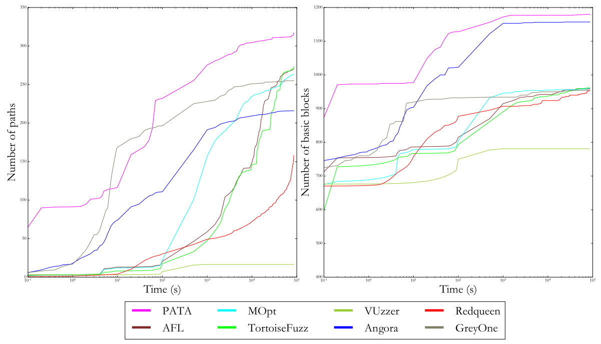

Fig. 4. The growing trend of the number of paths executed (left) and basic blocks covered (right) when fuzzing png2pnm by PATA, AFL, MOPT, TortoriseFuzz, VUzzer, Angora, REDQUEEN, and GREYONE over 5 runs in 24 hours. Displayed are the median and the 95% confidence intervals.

E. Effectiveness of path-aware algorithms

To evaluate the effectiveness of path-aware taint analysis, we implemented PATA-unaware, a weaker version of PATA with path-aware algorithms disabled. Note that path-aware taint analysis provides fundamental information for the path-oriented mutation such as byte position; the tightly-coupled nature requires us to disable them altogether. We evaluated PATA and PATA-unaware in Google's fuzzer-test-suite 24 hours for 5 times.

Table VI shows the average results for the number of paths, basic blocks, and bugs. It illustrates that path-awareness helps PATA execute 71% more paths, cover 20% more branches, and find 210% more bugs than the path-unaware version. Path-awareness enables precise analysis and efficient mutation. When analyzing the critical bytes, path changes in byte-level mutation can be detected to prevent under-tainting, and

TABLE VI
NUMBER OF PATHS, BASIC BLOCKS, AND UNIQUE BUGS OVER 5 RUNS IN 24 HOURS FOUND BY PATA-UNAWARE AND PATA

Project	Number of Paths		Number of Basic Blocks		Number of Bugs	
	PATA-unaware	PATA	PATA-unaware	PATA	PATA-unaware	PATA
boringssl	646	662	2263	2269	0	0
c-ares	35	34	66	66	1	1
freetype2	4453	9045	11120	15325	0	0
guetzli	1824	2006	6284	6290	1	1
harfbuzz	5336	5578	9896	9924	0	0
json	992	995	1607	1607	2	2
lcms	211	775	1658	2501	0	1
libarchive	2960	3752	4474	6295	0	0
libjpeg	2047	2643	2694	2822	0	0
libpng	573	628	1204	1413	0	0
libssh	24	484	895	1611	0	0
libxml2	1375	6101	4296	8243	1	3
llvm-libcxxabi	6735	6753	3913	3952	1	7
openssl-1.0.1f	407	408	4491	5094	1	1
openssl-1.0.2d	839	851	1389	1394	0	1
openssl-bignum	388	388	1111	1113	0	0
openssl-x509	923	966	5348	5349	0	0
openthread	777	766	5642	5619	0	0
pcre2	4637	19318	5591	9494	1	12
proj4	110	720	204	1752	0	0
re2	2813	2811	5647	5663	1	1
sqlite	303	300	1808	1808	0	0
vorbis	698	1270	1946	2071	0	0
woff2	717	827	2980	3061	1	1
wpantund	868	1610	11243	13019	0	0
Total	40691	69691	97770	117755	10	31
Improvement	–	71%↑	–	20%↑	–	210%↑

multiple occurrences of one constraint can be identified to prevent over-tainting. With more precise taint inference than conventional fuzzers, PATA mutates efficiently and explores each possible undiscovered state along the path.

F. Overhead of path-awareness

Figure 5 shows the time increase ratio of analyzing one input seed for path-aware and unaware taint analysis. It shows that path-awareness introduces about 38% of runtime overhead on average. The phenomenon can be demonstrated by the positive correlation between analysis time and overall results. For example, PATA uses about 64% more time to analyze a seed for `pcre2`, but the more precise analysis brings about 70% more basic blocks. Moreover, the path-aware taint analysis is only invoked when a new input seed is found. So it is valuable to conduct a precise analysis.

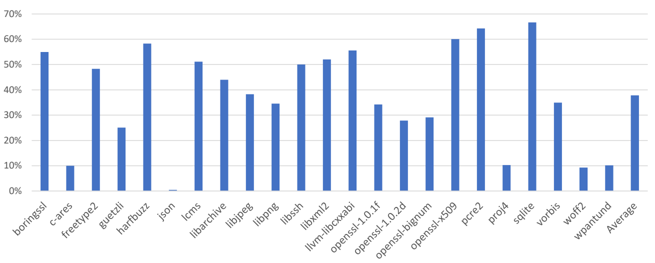

Fig. 5. The increment in the duration of analyzing one input speed brought by the path-aware algorithm.

Figure 6 shows the breakdown of time consumption on the three steps of PATA for processing a new input seed. On average, the time consumption of Step 1 (constraint variable collection), Step 2 (critical byte identification), and Step 3 (path-oriented mutation) account for about 31%, 55%, and 14%, respectively. Constraint variable collection is conducted before fuzzing. It has a sublinear relationship regarding the total instructions of the program. Critical byte identification is only invoked when a new input seed is discovered. It has a linear relationship regarding the input length. Based on the more accurate critical bytes analyzed by path-aware taint analysis, path-oriented mutation mutates input more precisely and spends only about 14% of the time.

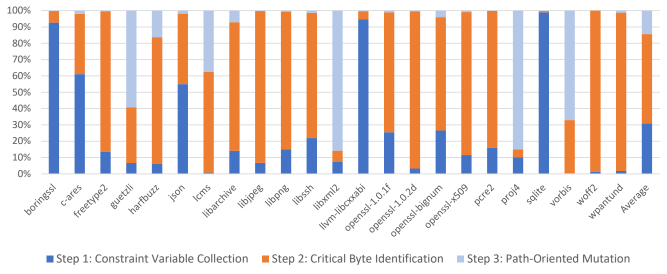

Fig. 6. The time consumption of Step 1: Constraint Variable Collection, Step 2: Critical Byte Identification, and Step 3: Path-Oriented Mutation.

Figure 7 presents the length of tracked and modified bytes compared to the length of the original input. The blue region represents the bytes that need to be modified and the orange represents others. On average, the proportion of the critical bytes for each constraint is about 3% of the input length, which means only a small part of the inputs is modified by PATA. As a result, although distinguishing paths does introduce overhead, it improves the precision of the mutation and overall performance.

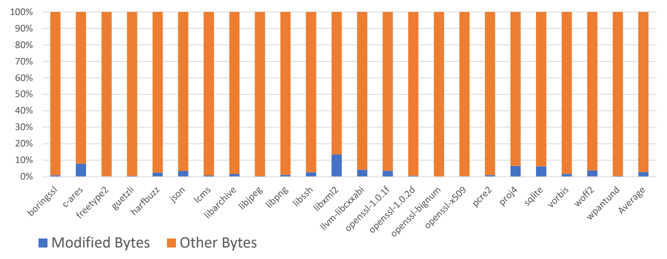

Fig. 7. The length of tracked and modified bytes compared to the length of the original input in projects of Google's fuzzer-test-suite.

VI. DISCUSSION

Our approach demonstrates good performance and effectiveness. However, there are still some potential factors that might limit the use of PATA in practice. First, an input might execute a long path. For some large projects, collecting paths may be resource-prohibitive. To overcome the problem, PATA could be configured in a more flexible way. For example, we can only record the constraint variables that have uncovered branches. In this way, all the monitored variables are meant to improve coverage. Second, the inputs might be very long. Long inputs may cost a lot of time for PATA to analyze the critical bytes. PATA mitigates this problem in three aspects. (1) PATA trims the input before analysis to reduce the input length but maintain its coverage. (2) PATA prefers short inputs. Long inputs will get processed with a low priority. (3) The mutation of PATA will not expand the length of existing inputs unless their lengths are related to some constraints. Finally, some constraints might be too hard for fuzzers to bypass by mutating but appear many times. Repeated mutation of these constraints costs resources but yields no gains. PATA adds their constraint variables to a blacklist to skip them when they are not solved many times.

The complex constraints along the path hinder the fuzzer from exploring more program states. The complexity can be explained by 1) the length of critical bytes, and 2) the complexity of transformations of these bytes. First, PATA employs path-aware taint analysis to locate critical bytes accurately, reducing the length of critical bytes caused by over-tainting. Second, PATA obtains more data flow features and tries to utilize them to explore each unexplored state along the path. For direct copies (17% of all constraint occurrences as Figure 10 shows), PATA's fast path handles them directly. For other cases, PATA could still handle them with linear search. However, some constraints might be still too hard to address. With limited critical bytes, combining symbolic execution [9, 10, 12, 21] like SAFL [31], Driller [28], QSYM [35] and DeepFuzzer [23] might be a choice to solve them. Currently, PATA does not support binary fuzzing. Nevertheless, our solution is performed on LLVM bitcode. By utilizing some tools (like Mcsema [13]) that can transform binaries to LLVM bitcode, PATA could still work when only already compiled binaries are available.

In this paper, we implement a path-aware taint inference. Taint inference is more suitable for fuzzing because it is fast and scalable to most programs. Nevertheless, it is also possible to integrate path-awareness into taint propagation. Specifically, instead of accumulating labels, the method should save unique labels for each occurrence. However, taint labels may take up a large amount of space. Thus saving labels for all occurrences would create substantial overhead. For example, VUzzer represents taint labels with a compressed bit-set data structure where each bit corresponds to an input byte. The size of labels would be large when the input bytes have a complex pattern and cannot be effectively compressed. When labels of all occurrences are recorded, the consumption is rather large.

VII. Related Work

A. Taint Propagation Based Fuzzers

Taint propagation is leveraged by many fuzzers [32, 18, 19, 26, 11] to determine which part of the input should be modified. Some works focus on locating the bytes which are used in security-sensitive operations. TaintScope [32] and BuzzFuzz [18] both use taint tracking to identify the input bytes that are used in sensitive systems or library calls and then focus on modifying such bytes. Dowser [19] performs taint propagation to figure out which parts of the input influence memory access in the target location. Other works focus on using taint propagation to guide fuzzing mutation. VUzzer [26] concentrates on guiding fuzzing to pass magic value validations. It tracks branches that compare variables against constants to identify critical bytes. By mutating critical bytes with collected immediate values, VUzzer generates inputs that are more likely to satisfy the constraints. Angora [11] concentrates on guiding fuzzing to solve the constraints and expand branch coverage. It utilizes the byte-level taint analysis to locate the bytes which flow into branches. Then it mutates these bytes with a gradient descent algorithm.

Different from taint propagation-based fuzzers, PATA infers taints with path-awareness. PATA distinguishes different occurrences of one constraint, which will not over-taint because of loops or multiple calls. Based on accurate critical bytes and occurrence values along the path, PATA mutates inputs and explores program states more efficiently.

B. Taint Inference Based Fuzzers

Many fuzzers [34, 6, 17] apply various mutation-based techniques to infer taints and boost fuzzing. They infer the dependence between bytes and constraints based on program state changes when mutating certain bytes. Some works infer the dependence based on control flow changes. ProFuzzer [34] conducts byte-level mutation and monitors the path changes. It groups the bytes which have the same exception path to probe the input fields (e.g. raw data or off-size). Then ProFuzzer mutates each field to trigger crashes and get more coverage. REDQUEEN [6] focuses on solving magic values and checksums in fuzzing. It colorizes an input seed by replacing each input byte with random bytes as many as possible but reserves the execution path. Only monitoring control flow changes is coarse-grained, thus some recent works infer taints based on value changes. GREYONE [17] stores the values of variables used in path constraints in a bitmap. It mutates seeds and checks value changes to infer taints.

In this paper, we propose a path-aware taint analysis that infers taints based on fine-grained value changes. Compared to REDQUEEN which uses colorizing, PATA discovers the critical bytes for constraints with path-aware taint analysis directly. Thus PATA could identify direct copies more effectively. Different from GREYONE which only records one value for a variable in a bitmap, PATA distinguishes different occurrences of one variable and records the value of each occurrence. When the value of one occurrence is changed, PATA captures it to avoid under-tainting because the value of other occurrences may not change. Furthermore, besides monitoring value changes, PATA is also aware of path changes, which is useful to avoid taint mismatching. Based on the critical bytes inferred for each node along paths, PATA efficiently mutates these bytes to explore uncovered branches.

VIII. Conclusion

This paper presents PATA, a novel fuzzer that aims to increase program coverage with path-aware taint analysis. It first identifies variables used by constraints and collects RVSs by recording values of each occurrence of these variables. Then it infers critical bytes for each constraint variable occurrence in the RVS by analyzing runtime values. Finally, it employs a path-oriented mutation. Along the path of the input, it precisely mutates critical bytes of constraint variables by combining variable features and occurrence values to explore the uncovered states. PATA outperforms several state-of-the-art fuzzers on Google's fuzzer-test-suite and LAVA-M in both coverage and bug triggering. In more widespread tests on real-world projects, PATA finds 40 unknown bugs, with 12 of them confirmed as CVEs.

IX. ACKNOWLEDGEMENTS

This research is sponsored in part by the NSFC Program (No. 62022046, U1911401, 61802180), National Key Research and Development Project (Grant No. 2019YFB1706203, No.2020AAA0107700), Key Research and Development Project in JiangXi Province (Grant No.20171ACE50025), NSFC Program in JiangSu Province (No.BK20180421).

References

[1] Microsoft Security Risk Detection (”Project Springfield”). https://www.microsoft.com/en-us/research/project/project-springfield/, 2015. [Online; accessed 26-January-2018].

[2] Continuous fuzzing for open source software. https://opensource.googleblog.com/2016/12/announcing-oss-fuzz-continuous-fuzzing.html, 2016. [Online; accessed 26-January-2018].

[3] libFuzzer in Chrome. https://chromium.googlesource.com/chromium/src/+/master/testing/libfuzzer/README.md, 2017. [Online; accessed 12-November-2017].

[4] DataFlowSanitizer. https://clang.llvm.org/docs/DataFlowSanitizer.html, 2020. [Online; accessed 27-July-2020].

[5] Abhishek Arya, Oliver Chang, Max Moroz, Martin Barbella, and Jonathan Metzman. Open sourcing clusterfuzz. https://opensource.googleblog.com/2019/02/open-sourcing-clusterfuzz.html, 2019.

[6] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. Redqueen: Fuzzing with input-to-state correspondence. In *NDSS*, volume 19, pages 1–15, 2019.

[7] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as markov chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1032–1043. ACM, 2016.

[8] Foster Brereton. Binspector: Evolving a security tool. https://blogs.adobe.com/security/2015/05/binspector-evolving-a-security-tool.html, 2015.

[9] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *OSDI*, volume 8, pages 209–224, 2008.

[10] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. Unleashing mayhem on binary code. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 380–394. IEEE, 2012.

[11] Peng Chen and Hao Chen. Angora: Efficient fuzzing by principled search. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 711–725. IEEE, 2018.

[12] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. S2E: A platform for in-vivo multi-path analysis of software systems. *ACM SIGPLAN Notices*, 46(3):265–278, 2011.

[13] Artem Dinaburg and Andrew Ruef. Mcsema: Static translation of x86 instructions to llvm. In *ReCon 2014 Conference, Montreal, Canada*, 2014.

[14] Joe W Duran and Simeon Ntafos. A report on random testing. In *Proceedings of the 5th international conference on Software engineering*, pages 179–183. IEEE Press, 1981.

[15] Michael Eddington. Peach fuzzing platform. *Peach Fuzzer*, page 34, 2011.

[16] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. Afl++: Combining incremental steps of fuzzing research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*, 2020.

[17] Shuitao Gan, Chao Zhang, Peng Chen, Bodong Zhao, Xiaojun Qin, Dong Wu, and Zuoning Chen. Greyone: Data flow sensitive fuzzing. In *29th USENIX Security Symposium (USENIX Security 20). USENIX Association, Boston, MA. https://www. usenix. org/conference/usenixsecurity20/presentation/gan*, 2020.

[18] Vijay Ganesh, Tim Leek, and Martin Rinard. Taint-based directed whitebox fuzzing. In *Proceedings of the 31st International Conference on Software Engineering*, pages 474–484. IEEE Computer Society, 2009.

[19] Istvan Haller, Asia Slowinska, Matthias Neugschwandtner, and Herbert Bos. Dowsing for Overflows: A Guided Fuzzer to Find Buffer Boundary Violations. In *USENIX Security Symposium*, pages 49–64, 2013.

[20] Sam Hocevar. zzuf - multi-purpose fuzzer. http://caca.zoy.org/wiki/zzuf, 2007. [Online; accessed 26-January-2018].

[21] James C King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[22] Caroline Lemieux and Koushik Sen. Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 475–485. ACM, 2018.

[23] Jie Liang, Yu Jiang, Mingzhe Wang, Xun Jiao, Yuanliang Chen, Houbing Song, and Kim-Kwang Raymond Choo. Deepfuzzer: Accelerated deep greybox fuzzing. *IEEE Transactions on Dependable and Secure Computing*, 2019.

[24] Chenyang Lyu, Shouling Ji, Chao Zhang, Yuwei Li, Wei-Han Lee, Yu Song, and Raheem Beyah. Mopt: Optimized mutation scheduling for fuzzers. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1949–1966, 2019.

[25] Barton P Miller, Louis Fredriksen, and Bryan So. An empirical study of the reliability of UNIX utilities. *Communications of the ACM*, 33(12):32–44, 1990.

[26] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. Vuzzer: Application-aware evolutionary fuzzing. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2017.

[27] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. Addresssanitizer: A fast address sanity checker. In Gernot Heiser and Wilson C. Hsieh, editors, *2012 USENIX Annual Technical Conference, Boston, MA, USA, June 13-15, 2012*, pages 309–318. USENIX Association, 2012. URL https://www.usenix.org/conference/atc12/technical-sessions/presentation/serebryany.

[28] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting Fuzzing Through Selective Symbolic Execution. In *NDSS*, volume 16, pages 1–16, 2016.

[29] Ari Takanen, Jared D Demott, and Charles Miller. *Fuzzing for software security testing and quality assur-*

ance. Artech House, 2008.

[30] Dmitry Vyukov. syzkaller is an unsupervised coverage-guided kernel fuzzer. https://github.com/google/syzkaller, 2015.

[31] Mingzhe Wang, Jie Liang, Yuanliang Chen, Yu Jiang, Xun Jiao, Han Liu, Xibin Zhao, and Jiaguang Sun. Safl: increasing and accelerating testing coverage with symbolic execution and guided fuzzing. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, pages 61–64. ACM, 2018.

[32] Tielei Wang, Tao Wei, Guofei Gu, and Wei Zou. TaintScope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection. In *Security and privacy (SP), 2010 IEEE symposium on*, pages 497–512. IEEE, 2010.

[33] Yanhao Wang, Xiangkun Jia, Yuwei Liu, Kyle Zeng, Tiffany Bao, Dinghao Wu, and Purui Su. Not all coverage measurements are equal: Fuzzing by coverage accounting for input prioritization.

[34] Wei You, Xueqiang Wang, Shiqing Ma, Jianjun Huang, Xiangyu Zhang, XiaoFeng Wang, and Bin Liang. ProFuzzer: On-the-fly Input Type Probing for Better Zero-Day Vulnerability Discovery. In *ProFuzzer: On-the-fly Input Type Probing for Better Zero-Day Vulnerability Discovery*, page 0. IEEE, 2019.

[35] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. QSYM: A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 745–761, 2018.

[36] Michal Zalewski. American fuzzy lop, 2015.

Appendix A
Additional Experiment Results

A. *The over-tainting issue of traditional taint analysis without path-awareness*

When the same variable is visited multiple times, the path-unaware analysis might produce the union of all occurrences as the result (over-tainting). To evaluate the extent of the phenomenon, we collect the seeds generated by AFL in 24 hours, then present the number of occurrences of constraint variables in Figure 8. It shows that constraint variables tend to occur multiple times in all projects, and some even reach the order of magnitude of 10^7. For fuzzers using propagation-based taint analysis without path-awareness, they may perform well on LAVA-M, but they may have trouble in maintaining their advantages on more complicated projects due to their lack of path-awareness. In contrast, PATA is aware of the paths and distinguishes between multiple occurrences of a constraint variable. Therefore, it performs well on real-world projects.

B. *The average number of unique path changes and the size of constraint set*

Figure 9 presents the average number of unique path changes when mutating an input. It also shows the size of the constraint set for the original path. In total, PATA achieves a 73% effective mutation rate. Specifically, the blue bar shows the average number of unique path changes when mutating critical bytes to pass constraints, while the orange bar shows the size of the constraint set. The ratio of the former to the latter reflects the effective rate of the mutation. The figure also demonstrates the benefit of path-awareness: for 9 of all listed projects, the number of path changes is even higher than the size of the constraint set. The unusual phenomenon indicates that PATA successfully solves more occurrences than the number of constraints. In other words, path-awareness allows detecting multiple occurrences of a constraint, which creates more opportunities for PATA's mutator.

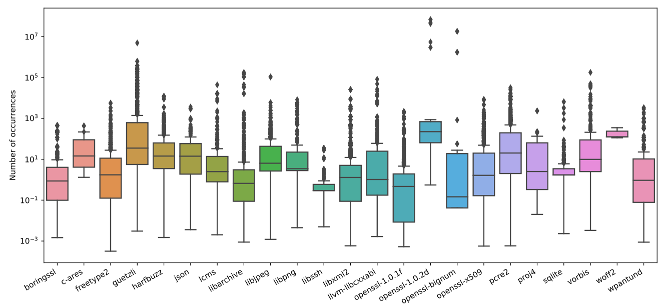

Fig. 8. The average number of occurrences for all constraint variables.

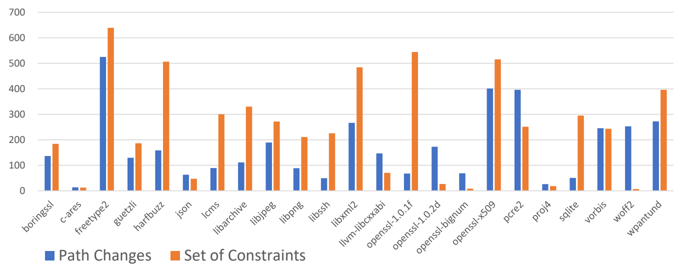

Fig. 9. The average number of unique path changes and the size of constraints set for each path.

C. *The occurrences of constraint variables which are related with direct copies*

Figure 10 shows that only 17% of all occurrences of constraint variables are related with direct copies on average. PATA can handle both direct copies and other cases. For direct copies, PATA's fast path directly handles them as prior works have done. For the most common cases where values are used after complex transforms, PATA can still handle them by linear search with operand feature (operand data type and length), pattern feature (constraint category, e.g. cmp), and runtime value; searching is fast because the search space is greatly reduced with precise path-aware taint analysis.

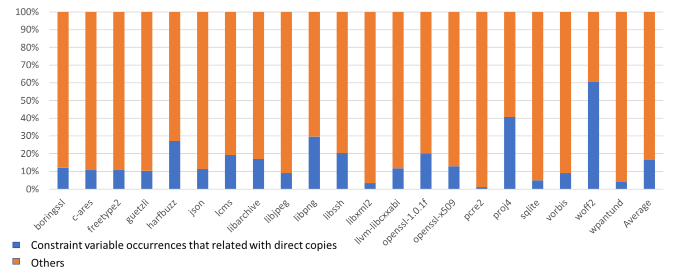

Fig. 10. The occurrences of all constraint variables that are related to direct copies only account for a small part.

D. Coverage of various fuzzers on LAVA-M

Table VII and VIII present the number of paths and basic blocks covered by fuzzing four projects in LAVA-M. They show that PATA performs well in four projects. Specifically, PATA executes 149%, 126%, 156%, 446%, 146%, 32%, and 6% more paths, finds 68%, 68%, 68%, 67%, 12%, 7%, and 56% more basic blocks than AFL, MOpt, TortoriseFuzz, VUzzer, Angora, Redqueen, and GreyOne, respectively.

TABLE VII
Average number of paths executed over 5 runs in 24 hours

Project	AFL	MOpt	TortoiseFuzz	VUzzer	Angora	Redqueen	GreyOne	PATA
base64	83	87	86	30	54	132	125	149
md5sum	167	183	158	69	107	229	186	179
uniq	32	36	29	33	19	37	52	60
who	62	74	62	25	168	253	444	469
Total	344	380	335	157	348	651	807	857
Improvement	149%↑	126%↑	156%↑	446%↑	146%↑	32%↑	6%↑	–

TABLE VIII
Average number of blocks covered over 5 runs in 24 hours

Project	AFL	MOpt	TortoiseFuzz	VUzzer	Angora	Redqueen	GreyOne	PATA
base64	299	301	298	320	378	380	315	385
md5sum	419	434	427	421	560	443	441	555
uniq	209	213	211	238	262	253	215	263
who	4495	4502	4495	4495	6924	7449	4890	7929
Total	5422	5450	5431	5474	8124	8525	5861	9132
Improvement	68%↑	68%↑	68%↑	67%↑	12%↑	7%↑	56%↑	–

AFL, MOpt, and TortoriseFuzz are greybox fuzzers. Their limited knowledge about the target projects restricts their coverage. VUzzer uses the taint analysis to locate magic bytes, however, it performs little better than AFL in block covering because of the inaccurate taint results. Angora, Redqueen, and GreyOne have higher coverage than VUzzer because they could solve some of the other constraints more than magic bytes. However, the inaccuracy of taint analysis also limits their effectiveness. On the contrary, PATA records the values along the execution path as a sequence. The ordering information not only supports multi occurrences of the same variable, but also enables recovery if the mutated input diverges the path. Thus PATA could perform better.

E. Number of bugs detected by various fuzzers on some real-world programs

Table IX presents the number of bugs detected by each fuzzer within 24 hours on more real-world programs from GitHub. It shows that PATA finds all 40 bugs, while other fuzzers only find a proportion of these bugs. Compared to AFL, MOpt, TortoriseFuzz, VUzzer, Angora, Redqueen, and GreyOne, PATA detects 20, 17, 21, 30, 23, 20, and 21 more unique bugs, respectively.

TABLE IX
Number of bugs detected by each fuzzer within 24 hours on some real-world programs

Project	AFL	MOpt	TortoiseFuzz	VUzzer	Angora	Redqueen	GreyOne	PATA
bigint	1	1	1	1	1	1	0	1
bitmap	1	1	1	1	1	1	1	1
cyclonedds	4	5	3	2	2	4	4	9
ffjpeg	1	1	1	1	1	1	1	1
genann	1	1	1	1	1	1	1	2
jpeg_encoder	1	1	1	0	1	1	1	3
jpeg-compressor	5	5	5	1	5	6	4	6
json	1	1	1	1	1	1	0	1
libconfig	1	2	1	0	0	1	1	3
libucl	1	1	1	0	1	1	1	1
libpng	0	0	0	0	0	0	0	3
lightmatrix	1	1	1	1	1	1	1	3
mxml	2	2	2	0	2	0	2	3
pdfalto	0	1	0	0	0	0	1	1
SmallerC	0	0	0	1	0	1	1	1
xml2json	0	0	0	0	0	0	0	1
Total	20	23	19	10	17	20	19	40
Increase	20↑	17↑	21↑	30↑	23↑	20↑	21↑	–

F. The performance of various fuzzers in 60 hours

Table X, Table XI, and Table XII show the coverage of each fuzzer together with their bug finding on fuzzing projects of Google's fuzzer-test-suite for 60 hours. They illustrate that PATA also outperforms other fuzzers. Specifically, PATA found 101%, 68%, 132%, 2209%, 390%, 66%, and 49% more paths, covered 21%, 17%, 29%, 101%, 49%, 14%, and 20% more basic blocks, and triggers 14, 9, 13, 33, 31, 16, and 24 more bugs than AFL, MOpt, TortoriseFuzz, VUzzer, Angora, Redqueen, and GreyOne, respectively.

TABLE X
Number of paths in 60 hours

Project	AFL	MOpt	TortoiseFuzz	VUzzer	Angora	Redqueen	GreyOne	PATA
boringssl	440	473	406	127	273	428	467	639
c-ares	27	26	26	26	19	29	33	35
freetype2	6038	7795	4560	68	2948	8184	9288	13599
guetzli	1038	1412	945	35	1012	932	1409	2224
harfbuzz	4631	5009	2865	97	1842	4609	2986	6323
json	688	703	632	41	295	565	818	1052
lcms	238	249	224	9	403	517	275	712
libarchive	1146	2143	1351	113	1239	1659	1635	4748
libjpeg	1124	1666	1009	40	739	1066	1263	2757
libpng	338	345	341	20	292	363	464	647
libssh	17	16	18	16	15	352	24	461
libxml2	1572	2164	2044	229	783	2657	3008	6853
llvm-libcxxabi	2700	4693	5112	220	854	5172	5666	6826
openssl-1.0.1f	421	308	245	14	35	251	360	931
openssl-1.0.2d	749	741	749	399	417	844	861	971
openssl-bignum	332	339	322	338	452	348	384	396
openssl-x509	864	875	852	865	874	866	939	980
openthread	522	496	527	43	346	911	835	857
pcre2	14179	15072	9480	335	2209	14647	16728	19477
proj4	64	70	66	22	125	51	124	4983
re2	2016	1678	1950	307	995	1611	2880	2957
sqlite	225	235	233	46	168	235	288	302
vorbis	641	712	727	64	563	849	916	1341
woff2	554	642	563	17	5	536	897	870
wpantund	1158	1966	976	144	233	2990	3685	2980
Total	41722	49828	36223	3635	17136	50672	56233	83921
Improvement	101%↑	68%↑	132%↑	2209%↑	390%↑	66%↑	49%↑	–

TABLE XI
Number of basic blocks in 60 hours

Project	AFL	MOpt	TortoiseFuzz	VUzzer	Angora	Redqueen	GreyOne	PATA
boringssl	2269	2269	2205	2185	2201	2267	2208	2267
c-ares	64	64	63	66	63	64	66	66
freetype2	12355	12994	11428	5670	10607	15811	13338	15989
guetzli	6183	6325	6131	5391	6231	6237	6180	6340
harfbuzz	9760	10052	8925	5246	8397	9931	8875	10174
json	1578	1580	1532	641	1483	1590	1577	1607
lcms	1667	1655	1687	643	2403	2237	1810	2531
libarchive	3481	4299	3276	1351	5128	3604	3196	6950
libjpeg	2369	2821	2340	1220	2359	2456	2383	2828
libpng	1182	1187	1180	682	1330	1393	1195	1418
libssh	893	893	893	893	817	1664	908	1589
libxml2	4543	5379	4590	2803	3854	5290	4898	9105
llvm-libcxxabi	3618	3920	3669	1865	3082	3874	3722	3957
openssl-1.0.1f	6351	5677	3077	885	1179	4045	5617	6417
openssl-1.0.2d	1416	1412	1408	1369	1096	1408	1402	1412
openssl-bignum	1101	1099	1097	1110	1104	1102	1111	1115
openssl-x509	5304	5304	5281	5281	5329	5304	5570	5572
openthread	5645	5630	5676	3245	5371	7518	5678	5726
pcre2	9316	9466	8050	2551	4197	9659	9300	9879
proj4	204	204	196	126	669	196	210	5342
re2	5605	5593	5590	4177	4819	5593	5662	5701
sqlite	1805	1805	1803	1377	1442	1805	1808	1808
vorbis	2046	2065	2024	1298	1910	2053	1970	2104
woff2	3016	3031	3005	2556	164	2985	3100	3104
wpantund	12754	13632	12860	10442	9594	13540	13557	13790
Total	104525	108356	97986	63073	84829	111626	105341	126791
Improvement	21%↑	17%↑	29%↑	101%↑	49%↑	14%↑	20%↑	–

TABLE XII
Number of Bugs in 60 hours

Project	AFL	MOpt	TortoiseFuzz	VUzzer	Angora	Redqueen	GreyOne	PATA
c-ares	1	1	1	0	1	1	1	1
guetzli	1	1	1	0	0	0	0	2
json	2	2	2	0	0	2	1	2
lcms	1	1	1	0	0	0	0	1
libxml2	2	2	1	0	0	2	0	3
llvm-libcxxabi	5	6	4	0	0	5	4	7
openssl-1.0.1f	1	1	1	0	1	1	0	1
openssl-1.0.2d	1	1	1	0	0	1	0	1
pcre2	4	7	6	0	0	4	3	12
re2	0	1	1	0	0	1	0	2
woff2	1	1	1	0	0	0	0	1
Total	19	24	20	0	2	17	9	33
Increase	14↑	9↑	13↑	33↑	31↑	16↑	24↑	–

Listing 5 shows one bug of `libxml2` which is only found by PATA. The bug will be triggered when `tlen` exceeds the length of `ctxt->input`. The bug is very difficult to find because it has many preconditions. For instance, the condition in Line 2 (logical operation), Line 7 (byte comparison), Line 10 (logical operation), Line 11 (integer comparison), and Line 12 (byte-array comparison) should all be satisfied. In particular, the constraint variables related to these conditions all occur multiple times because of the loop and the recursive call. It is difficult for other fuzzers to detect bugs in such deep paths without the path-aware taint analysis of PATA. More cases could also be found in the PATA's website.

```
1  void xmlParseContent(xmlParserCtxtPtr ctxt) {
2    while ((RAW != 0) && ((RAW != '<') || /*...*/)) {
3      //...
4      if ((*cur == '<') && (cur[1] == '?')) {
5        ///...
6      }
7      else if (*cur == '<') {
8        if(/*...*/) { return; }
9        xmlParseContent(ctxt);
10       if (ctxt->sax2) {
11         if ((tlen > 0) &&
12         (xmlStrncmp(ctxt->input,ctxt->name,tlen)==0)) {
13           if (ctxt->input[tlen] == '>') // <= OVERFLOW
14           //...
15         }
16       }
17     }
18     //...
19   }
20 }
```

Listing 5. A buffer overflow detected by PATA. The program must pass through the constraints which occur multiple times in Line 2 (logical operation), Line 7 (byte comparison), Line 10 (logical operation), Line 11 (integer comparison), and Line 12 (byte-array comparison).

G. Mann-Whitney U Test Results

We repeat all the 24-hour experiments 5 times to avoid the influence of random variations. Table XIII shows the p-value of the Mann-Whitney U test on the number of paths for projects in Google's fuzzer-test-suite between PATA and AFL, MOPT, TortoriseFuzz, VUzzer, Angora, REDQUEEN, and GREYONE. It shows that for most of the projects, the differences are significant at $p < 0.05$.

TABLE XIII
P-VALUES OF THE MANN-WHITNEY U TEST ON THE PATHS OVER 5 RUNS IN 24 HOURS

Project	AFL	MOPT	TortoiseFuzz	VUzzer	Angora	REDQUEEN	GREYONE
boringssl	0.006	0.006	0.006	0.006	0.006	0.006	0.006
c-ares	0.005	0.005	0.005	0.004	0.005	0.004	0.079
freetype2	0.006	0.006	0.006	0.006	0.006	0.006	0.105
guetzli	0.006	0.006	0.006	0.006	0.006	0.006	0.006
harfbuzz	0.006	0.006	0.006	0.006	0.006	0.006	0.006
json	0.006	0.006	0.006	0.006	0.006	0.006	0.006
lcms	0.006	0.006	0.006	0.006	0.006	0.006	0.006
libarchive	0.006	0.006	0.006	0.006	0.006	0.006	0.006
libjpeg	0.006	0.006	0.006	0.006	0.006	0.006	0.006
libpng	0.006	0.006	0.006	0.006	0.006	0.006	0.006
libssh	0.006	0.006	0.006	0.006	0.006	0.006	0.004
libxml2	0.006	0.006	0.006	0.006	0.006	0.006	0.006
llvm-libcxxabi	0.006	0.006	0.006	0.006	0.006	0.006	0.006
openssl-1.0.1f	0.006	0.018	0.006	0.006	0.006	0.006	0.030
openssl-1.0.2d	0.006	0.006	0.006	0.006	0.006	0.105	0.018
openssl-bignum	0.006	0.006	0.006	0.338	0.057	0.006	0.064
openssl-x509	0.006	0.006	0.006	0.005	0.006	0.005	0.006
openthread	0.006	0.006	0.006	0.006	0.006	0.072	0.011
pcre2	0.006	0.006	0.006	0.006	0.006	0.006	0.006
proj4	0.006	0.006	0.006	0.006	0.500	0.265	0.006
re2	0.006	0.006	0.006	0.006	0.006	0.006	0.202
sqlite	0.006	0.006	0.006	0.006	0.006	0.006	0.006
vorbis	0.006	0.006	0.006	0.006	0.006	0.006	0.006
woff2	0.006	0.006	0.006	0.006	0.006	0.006	0.265
wpantund	0.006	0.018	0.006	0.006	0.006	0.417	0.417

Table XIV shows the p-value of the Mann-Whitney U test on the number of basic blocks for projects in Google's fuzzer-test-suite between PATA and AFL, MOPT, TortoriseFuzz, VUzzer, Angora, REDQUEEN, and GREYONE. It also shows that for most of the cases, the differences are significant at $p < 0.05$.

TABLE XIV
P-VALUES OF THE MANN-WHITNEY U TEST ON THE BASIC BLOCKS OVER 5 RUNS IN 24 HOURS

Project	AFL	MOPT	TortoiseFuzz	VUzzer	Angora	REDQUEEN	GREYONE
boringssl	0.004	0.115	0.005	0.005	0.005	0.016	0.005
c-ares	0.003	0.003	0.002	0.033	0.003	0.002	0.002
freetype2	0.006	0.006	0.006	0.006	0.006	0.006	0.006
guetzli	0.006	0.458	0.006	0.006	0.006	0.006	0.006
harfbuzz	0.006	0.030	0.006	0.006	0.006	0.006	0.006
json	0.004	0.004	0.004	0.004	0.004	0.004	0.004
lcms	0.006	0.006	0.005	0.006	0.047	0.006	0.006
libarchive	0.006	0.006	0.006	0.006	0.006	0.006	0.006
libjpeg	0.006	0.018	0.006	0.006	0.006	0.006	0.006
libpng	0.005	0.005	0.006	0.005	0.005	0.005	0.004
libssh	0.005	0.005	0.005	0.005	0.005	0.338	0.004
libxml2	0.006	0.006	0.006	0.006	0.006	0.006	0.006
llvm-libcxxabi	0.006	0.006	0.006	0.006	0.006	0.006	0.006
openssl-1.0.1f	0.338	0.202	0.265	0.006	0.006	0.500	0.006
openssl-1.0.2d	0.007	0.006	0.263	0.006	0.005	0.008	0.028
openssl-bignum	0.014	0.050	0.005	0.003	0.224	0.005	0.033
openssl-x509	0.002	0.003	0.002	0.002	0.003	0.003	0.003
openthread	0.047	0.201	0.018	0.006	0.006	0.006	0.500
pcre2	0.006	0.006	0.006	0.006	0.006	0.265	0.006
proj4	0.004	0.005	0.062	0.006	0.500	0.262	0.038
re2	0.006	0.006	0.006	0.006	0.006	0.006	0.298
sqlite	0.003	0.003	0.004	0.004	0.004	0.003	0.089
vorbis	0.037	0.071	0.006	0.006	0.006	0.006	0.008
woff2	0.006	0.006	0.005	0.006	0.006	0.006	0.373
wpantund	0.006	0.006	0.006	0.006	0.006	0.072	0.006