Steelix: Program-State Based Binary Fuzzing

Yuekang Li
School of Computer Science and Engineering
Nanyang Technological University
Singapore

Bihuan Chen*
School of Computer Science and
Shanghai Key Lab. of Data Science
Fudan University
China

Mahinthan Chandramohan
School of Computer Science and Engineering
Nanyang Technological University
Singapore

Shang-Wei Lin†
School of Computer Science and Engineering
Nanyang Technological University
Singapore

Yang Liu
School of Computer Science and Engineering
Nanyang Technological University
Singapore

Alwen Tiu
School of Computer Science and Engineering
Nanyang Technological University
Singapore

ABSTRACT

Coverage-based fuzzing is one of the most effective techniques to find vulnerabilities, bugs or crashes. However, existing techniques suffer from the difficulty in exercising the paths that are protected by magic bytes comparisons (e.g., string equality comparisons). Several approaches have been proposed to use heavy-weight program analysis to break through magic bytes comparisons, and hence are less scalable. In this paper, we propose a program-state based binary fuzzing approach, named Steelix, which improves the penetration power of a fuzzer at the cost of an acceptable slow down of the execution speed. In particular, we use light-weight static analysis and binary instrumentation to provide not only coverage information but also *comparison progress* information to a fuzzer. Such program state information informs a fuzzer about *where the magic bytes are located in the test input* and *how to perform mutations to match the magic bytes efficiently.* We have implemented Steelix and evaluated it on three datasets: LAVA-M dataset, DARPA CGC sample binaries and five real-life programs. The results show that Steelix has better code coverage and bug detection capability than the state-of-the-art fuzzers. Moreover, we found one CVE and nine new bugs.

CCS CONCEPTS

• **Security and privacy → Software security engineering**;

KEYWORDS

binary fuzzing, coverage-based fuzzing, binary instrumentation

ACM Reference format:
Yuekang Li, Bihuan Chen, Mahinthan Chandramohan, Shang-Wei Lin, Yang Liu, and Alwen Tiu. 2017. Steelix: Program-State Based Binary Fuzzing. In *Proceedings of 2017 11th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, Paderborn, Germany, September 4–8, 2017 (ESEC/FSE'17),* 11 pages.
https://doi.org/10.1145/3106237.3106295

1 INTRODUCTION

Since its introduction in early 1990s [37], fuzzing has become one of the most effective and scalable testing techniques to find vulnerabilities, bugs or crashes in commercial off-the-shelf (COTS) software. It has also been widely used by mainstream software companies such as Google [28], Microsoft [15] and Adobe [22] to ensure the quality of their software products. The key idea of fuzzing is to feed the program under test (PUT) with a large amount of malformed test inputs to trigger unintended program behaviors, such as crashes or hangs.

The existing fuzzing approaches can be classified by two dimensions. Based on how the structural knowledge of the PUT is utilized, fuzzers can be classified as white-box, black-box or grey-box. White-box fuzzers (e.g., [29, 30, 39]) either have access to the source code of the PUT, or rely on binary lifting [34] to translate assembly into an intermediate language. They usually apply heavy-weight program analysis such as symbolic execution [43] to improve the effectiveness but may have scalability problems. Black-box fuzzers (e.g., [49]) have no knowledge about the internals of the PUT and thus are less effective. Grey-box fuzzers (e.g., [8, 24]) are in between. They apply light-weight program analysis to extract partial information of the PUT without sacrificing the fast execution speed of tests.

On the other hand, based on how the test inputs to the PUT are generated, fuzzers can be classified as mutation-based or generation-based. Mutation-based fuzzers (e.g., [8, 23, 29]) start with a set of pre-provided test inputs (i.e., seeds), and generate new test inputs by mutating these test inputs (e.g., byte flipping). They are effective to fuzz programs that process compact and unstructured data formats (e.g., image). Generation-based fuzzers (e.g., [14, 39, 50]) start with no test inputs, and construct test inputs based on the knowledge of the input format or grammar. They are more suitable for programs that process highly-structured inputs (e.g., XML).

In this paper we focus on grey-box mutation-based fuzzing. One of the most successful techniques is coverage-based fuzzing, which uses light-weight instrumentation to extract coverage information for each executed test input in order to determine which test inputs

[*] Corresponding Author. Also with Nanyang Technological University, Singapore.
[†] Shang-Wei Lin and Yang Liu have equal contribution in this work.

ESEC/FSE'17, September 4–8, 2017, Paderborn, Germany
© 2017 Association for Computing Machinery.
ACM ISBN 978-1-4503-5105-8/17/09…$15.00
https://doi.org/10.1145/3106237.3106295

```c
int main(void) {
  char str[4];
  gets(str);
  if(strcmp(str, "MAZE") == 0)
    // trigger the crash
  return 0;
}
```

(a) Sample Code 1

```c
int main(void) {
  if (getchar() == 'M')
    if (getchar() == 'A')
      if(getchar() == 'Z')
        if(getchar() == 'E')
          // trigger the crash
  return 0;
}
```

(b) Sample Code 2

Figure 1: Motivation Examples

should be retained for fuzzing. Specifically, if a test input can trigger the execution of a new basic block, it is considered as interesting and retained; otherwise, it is discarded. Thus, coverage-based fuzzers explore execution paths of a PUT in an incremental manner. AFL [8] is the state-of-the-art coverage-based fuzzer, and has discovered hundreds of high-profile vulnerabilities [16].

However, coverage-based fuzzing has limited penetration power to exercise the paths protected by *magic bytes comparisons*. Magic bytes refer to the bytes in the test input which are uses in comparison instructions. For example, the string "MAZE" in the program in Fig. 1a is considered as four magic bytes. In this case, AFL has to mutate all the four magic bytes correctly at once to trigger the crash because mutating one, two or three bytes correctly cannot lead to new coverage. Thus, it is very difficult for AFL to trigger the crash; i.e., AFL needs at most 2^{4*8} executions of the program in Fig. 1a. The challenge is that coverage-based fuzzers do not have the knowledge of *where the magic bytes are located in the test input* and *how to perform mutations to match the magic bytes efficiently*.

Several advances have been already made to combine coverage-based fuzzing with some program analysis techniques to address the challenge. For example, Driller [44] uses concolic execution to solve those comparison constraints. VUzzer [40] uses dynamic taint analysis to penetrate those comparisons. AFL-lafintel [17] applies program transformation at LLVM IR level to convert a magic bytes comparison into multiple nested one-byte comparisons. Although they have shown promising results, both Driller and VUzzer rely on heavy-weight program analysis (i.e., concolic execution suffers from the infamous path explosion problem, and dynamic taint analysis can greatly slow down the execution speed); and AFL-lafintel works at the source code level and fails to know where the magic bytes are located in the test input.

In this paper, we propose a program-state based fuzzer Steelix[1], which works at the binary level and can exercise paths protected by magic bytes comparisons at the cost of an acceptable slow down of the execution speed. The key idea of Steelix is that we not only collect the coverage information but also collect the *comparison progress information* (i.e., whether more bytes are correctly matched in magic bytes comparisons). Thus program state in this paper refers to coverage and comparison progress. Whenever a test input, generated by mutating some byte, triggers new program states, we infer the location of the magic bytes as the mutated byte and its neighbors in the test input, and retain the test input for further fuzzing.

In particular, instead of relying on heavy-weight program analysis, Steelix leverages light-weight static analysis and binary instrumentation to collect the coverage and comparison progress information as the dynamic feedbacks to guide the mutation. Static analysis filters out uninteresting comparisons (e.g., one-byte comparisons), and extracts the information of interesting comparisons. Based on the extracted information, binary instrumentation instruments the PUT to obtain the actual value of comparison operands and generate comparison progress information during runtime. Then the fuzzer takes the instrumented PUT, and uses the collected feedbacks to perform adaptive mutation.

We have implemented the proposed approach by extending AFL, and evaluated the effectiveness of Steelix using two sets of widely-used benchmark programs (i.e., LAVA-M [27] and DARPA CGC sample binaries [9]) and five real-life programs (i.e., `pngfix`, `tcpdump`, `tiffcp`, `tiff2pdf` and `gzip`). Steelix outperformed both VUzzer [40] and AFL-lafintel [17] in three out of the four binaries from LAVA-M and found an average of 3× more bugs. Moreover, Steelix covered an average of respectively 12.7%, 9.7% and 14.2% more lines of code, functions and branches than AFL-dyninst [11] in three out of the five real-life programs where magic bytes comparisons are very common, and found more bugs than AFL-dyninst in CGC sample binaries and real-life programs. Specifically, we found one CVE and nine new bugs, and three of them were not found by AFL-dyninst.

In summary, this work makes the following contributions.

- We proposed a program-state based binary fuzzing approach to exercise paths protected by magic bytes comparisons at the cost of an acceptable slow down of the execution speed.
- We proposed light-weight static analysis and binary instrumentation to collect both coverage and comparison progress information as the dynamic feedbacks to guide adaptive mutation.
- We implemented and evaluated Steelix on various benchmark and real-life programs, which showed promising results.
- We found one CVE and nine previously unknown bugs in some widely-used real-life programs.

2 OVERVIEW

In this section, we first introduce a motivating example, and then present an overview of the proposed approach.

2.1 Motivation Example

Coverage-based fuzzers, such as AFL, use the coverage information to determine which mutated test input should be kept. For example, for the program in Fig. 1b, given the test input "XXXX", AFL will keep the test input "MXXX" after mutating the first byte correctly because "MXXX" can pass the first `if` conditional and trigger new code coverage. Based on "MXXX", AFL can generate the test input "MAXX" that will also be kept. In this incremental way, AFL can eventually generate the test input "MAZE" and trigger the crash.

However, for the program in Fig. 1a that has the same logic as the program in Fig. 1b, AFL will have difficulty in triggering the crash because mutating one byte correctly does not trigger new code coverage. For example, given the test input "XXXX", AFL can generate "MXXX" after some mutations, but "MXXX" does not trigger new coverage. Thus, AFL will discard "MXXX" although some progress has been made to pass the magic bytes comparison. In this case, AFL has to mutate the whole four magic bytes correctly at once to trigger the crash. AFL needs at most 2^{4*8} executions to trigger the crash in Fig. 1a, but needs at most $4 * 2^8$ executions for Fig. 1b.

To break through the magic bytes comparison in Fig. 1a, AFL-lafintel [17] attempts to transform the program in Fig. 1a into the

[1] A Pokémon that can dig deep below the surface.

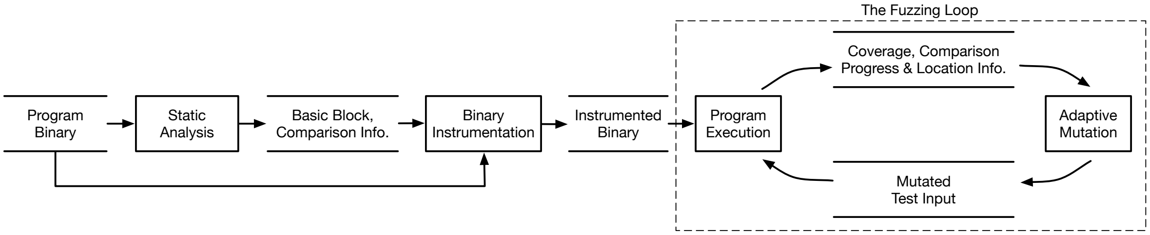


Figure 2: An Overview of the Proposed Approach Steelix

program in Fig. 1b, and relies on AFL's capability to incrementally trigger the crash. However, if the test input of the program in Fig. 1b contains 1000 bytes and the magic bytes are just four of them, which is quite common in real-life programs, AFL will still have difficulty in triggering the crash because it does not know where the magic bytes are located in the test input and thus cannot mutate the test input efficiently. On the other hand, Driller [44] and VUzzer [40] respectively use concolic execution and dynamic taint analysis to break through magic bytes comparisons. However, as concolic execution and dynamic taint analysis are often known as heavy-weight techniques, Driller and VUzzer can handle the program in Fig. 1a, but may suffer from scalability problems for real-life programs.

Following the previous examples, we have two observations. First, the test input "MXXX" has made some progress to match the magic bytes, and should be kept for further fuzzing. Second, "MXXX" is mutated from "XXXX", which shows that part of the magic bytes is located at the first byte, and may also be located at the neighbors of the first byte. Motivated by these observations, we propose to collect both coverage and comparison progress information, infer the location information of magic bytes by tracking if mutating certain byte can lead to new coverage or comparison progress, and use such information to guide the mutation.

2.2 Approach Overview

Our approach is designed to fuzz a PUT directly on its executable binary with two considerations. First, the source code of a PUT is not always available. By working at the binary level, Steelix can be applicable to both open and close source programs. Second, the comparison operands in assembly code lose their type information. The comparisons of integer, float or string/buffer become the comparisons of bytes. Hence, comparisons at assembly code level are more explicit for analysis than those at the source code level.

Fig. 2 gives an overview of Steelix, which contains three main components: *static analysis*, *binary instrumentation*, and *the fuzzing loop*. In particular, static analysis (see Section 3.1) takes the program binary as an input and disassembles it. Based on the assembly code, it filters out uninteresting comparisons according to several rules so that only a portion of the comparisons are dynamically analyzed during fuzzing. Then it extracts the information of those interesting comparisons and the information of basic blocks, which tells binary instrumentation *where to instrument* and *what to instrument*. Note that basic blocks are used to collect coverage information for the fuzzer, which are instrumented in the same way as current coverage-based fuzzing approaches (e.g., AFL). Thus, we will not discuss how to analyze and instrument basic blocks in Section 3.1 and 3.2.

The statically extracted information, together with the program binary, are then passed to binary instrumentation (see Section 3.2). The binary instrumentation has two main concerns. First, we need to mark comparison progress in a compact way as comparison progress is recorded in the shared memory whose size is designed to be limited (64KB) for efficiency [8]. Second, we instrument the program to get the actual value of comparison operands and generate comparison progress information during fuzzing.

Finally, the fuzzing loop (see Section 3.3) takes the instrumented binary and starts the fuzzing. Specifically, after executing the instrumented program, the fuzzer will get the coverage and comparison progress feedback, and derive the location information of the magic bytes based on the feedback. The coverage, comparison progress and location information are then used to guide the adaptive mutation, i.e., choosing suitable mutation operators.

3 METHODOLOGY

In this section, we elaborate each component in Fig. 2 in details.

3.1 Static Analysis

The purpose of static analysis in Steelix is to provide the basic block and comparison information for binary instrumentation (Section 3.2). Here we only discuss the comparison information.

3.1.1 Comparison Instructions. Static analysis first disassembles the program binary. The instruction set for assembly varies on different platforms. In this work, we focus on the x86 32-bit instruction set. The comparisons in x86 assembly can be achieved by using the `cmp`/`test` instructions or function calls.

Both `test` and `cmp` instructions have two operands. The `test` instruction performs a bitwise logical AND operation and sets the flags. For example, the instrument `test ebx, ebx` will set the Zero Flag (ZF) if the value of register `ebx` is `0`.

The `cmp` instruction subtracts the operands and set the flags. For example, the instrument `cmp dword ptr [ebp-4], 9` will set the ZF if the memory content of `ebp-4` is `9`.

The operands of `cmp` and `test` instructions can be a register, memory reference, or immediate value. The size of an operand can be 4 bytes (dword), 2 bytes (word), or 1 byte (byte).

For comparing strings or buffer values, the program first pushes the values of the function arguments onto the stack and then invokes the corresponding functions (e.g., `strcmp/strncmp`).

3.1.2 Filtering Out Uninteresting Comparisons. In Steelix, we only perform instrumentation on the *interesting* comparison instructions because the instrumentation slows down the program execution. We want to add them as precise as possible to reduce the execution overhead. Interesting `cmp`/`test` instructions are those whose comparison operands are meaningful for Steelix. The following rules describe how *uninteresting* instructions are filtered out.

```
int main(void) {
  int magic_number = 666;
  int in_num;
  scanf("%d", &in_num);
  if (in_num == magic_number)
    // trigger the crash
  return 0;
}
```

(a) Sample Code 3

```
int main(void) {
  int magic_number = 666;
  char in_str[20];
  int in_num;
  gets(in_str);
  in_num = hash(in_str);
  if (in_num == magic_number)
    // trigger the crash
  return 0;
}
```

(b) Sample Code 4

Figure 3: Examples of Comparisons

- *One-byte comparisons are not instrumented.* As discussed, the size of operands used in a `cmp/test` instruction can be 1, 2 or 4 bytes. One-byte comparisons can be easily matched with those default bit-flippings or arithmetic plus/minus mutations used by AFL.
- *Comparisons of function return values are not instrumented.* The reason is that the computations in functions can make the link between comparison operands and input bytes less explicit. For example, for the program in Fig. 3a, the comparison between `in_num` and `magic_number` is interesting as `in_num` is directly from the test input. However, the comparison in the program in Fig. 3b is uninteresting. This is because the comparison uses the result of a hash function that is linked with all bytes in the test input. To match one byte correctly at the comparison requires mutating many bytes in the test input, and the complexity is not reduced.

3.1.3 Extracting Comparison Information. After filtering out the *uninteresting* comparisons, Steelix extracts the information of the remaining comparisons with static analysis by scanning through the assembly and generating two lists of comparison information.

The first list keeps the information of the interesting `cmp` and `test` instructions. Each entry of this list is in the following format:

```
instruction_address: operand1_info: operand2_info
```

where `instruction_address` is the address of the instruction, and `operand#_info` holds the type of the operand, i.e., whether it is a register, a memory reference, or an immediate value. It also contains some other useful information for getting the actual value of the operand at runtime. For example, in the following statement, the offset `-4` is also included in `operand1_info`.

```
cmp dword ptr [ebp-4], 9
```

The second list keeps the information of function calls of `strcmp`, `strncmp` or `memcmp`. Each entry of this list contains the address of the function call and the name of the called function:

```
function_call_instruction_address: function_name
```

Note that the address of instructions in each entry informs the binary instrumentation of where to add the instrumentation.

3.2 Binary Instrumentation

Static analysis only provides static information of the interesting comparisons in the program binary, and the actual value of a comparison operand remains unknown during the static analysis unless it is an immediate value. Thus, based on the statically extracted comparison information, we adopt program instrumentation to provide runtime feedback (i.e., get the actual value of comparison operands and generate comparison progress information) for the fuzzer.

3.2.1 Comparison Progress. Comparison progress, together with the coverage information, is recorded in the shared memory whose size is designed to be limited (64KB) for efficiency [8]. As we record the progress for all comparisons, we need to mark the comparison progress in a compact way to fit with the shared memory. Here we define comparison progress as *how many consecutive bytes, starting from the first or last byte of the comparison operands, are matched.*

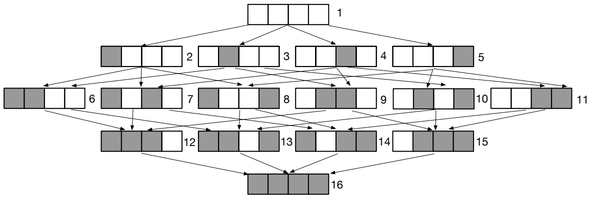


Figure 4: The States of a Four-Byte Comparison (A Shaded Block Means a Matched Byte)

For example, a four-byte comparison has 16 different states as shown in Fig. 4. It is too costly to keep the information of all the 16 states because the number of states will explode when the number of magic bytes grows. To avoid the state explosion, we selectively use some of the states to mark the progress of a comparison. From Fig. 4, we can see that there are 24 paths from state 1 to state 16. According to our definition of the comparison progress, we use the states on the path 1 → 2 → 6 → 12 → 16 and the states on the path 1 → 5 → 11 → 15 → 16 to infer the comparison progress.

The reason for choosing those two paths is that given any two consecutive states on those paths, we can know which exact byte the fuzzer should mutate in the next iteration. For example, given state 2 (with the first byte matched) and state 6 (with the first and second bytes matched), we can infer that we should mutate the next byte in the *forward* direction, i.e., the third byte. Similarly, given state 11 (with the last two bytes matched) and state 15 (with the last three bytes matched), we can infer that we should mutate the next byte in the *backward* direction, i.e., the first byte.

Furthermore, states on the same row in Fig. 4 can be merged into one situation. For example, both state 2 and 5 are matching one byte correctly and can be merged into one situation. The difference is that to make comparison progress, we need to mutate the next byte in forward direction for state 2 but in backward direction for state 5, and the direction of mutation can be inferred as discussed above.

Thus, instead of considering all the 16 states, we only consider 5 situations to represent the progress of a four-byte comparison: matching 0, 1, 2, 3, and 4 bytes. These 5 situations correspond to rows 1 to 5 in Fig. 4, respectively. Matching 0 byte means that there is no progress at all. Matching 4 bytes means that the magic bytes are found. The situations in between are *intermediate steps*. For example, if we have a test input at state 2, any mutation leading to new test inputs at states other than state 6 will not be counted as making progress and the fuzzer will focus on producing a test input at state 6. The detail of how the mutation operator utilizes the progress information will be discussed in Section 3.3. By categorizing comparison states into different situations, we can reduce the number of shared memory entries for a n-byte comparison from 2^n to $n + 1$.

3.2.2 Instrumentation Mechanisms. The instrumentations we add to the PUT are to generate the program state feedback, i.e., the coverage change and comparison progress, for the fuzzer. Here, we focus on the instrumentations for comparisons.

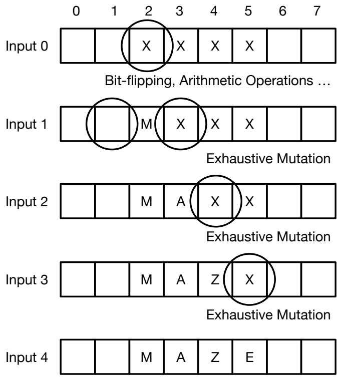


Figure 5: An Example of Adaptive Mutation

As introduced in Section 3.1.3, two lists of comparison information are generated by static analysis. One list keeps the information of the `cmp/test` instructions and the other one keeps the information of the comparison function calls. These two lists are used for providing guidance about where and what to instrument.

Specifically, for `cmp` and `test` instructions, we add the instrumentations before them based on the address information in the list. The logic flow inside the instrumentations is as follows.

- First, the actual values of comparison operands are extracted during runtime. For a register operand, its value can be directly accessed given the register name. For a memory reference operand, the memory address is computed and the operand value is generated by dereferencing the corresponding memory addresses.
- Then, the operand values are used for generating the comparison progress information as discussed in Section 3.2.1.
- Finally, the fuzzer is informed about the comparison progress information via the shared memory.

For the comparisons using function calls, since the function arguments can be hard to access from the stack, we replace the function calls with calls to our implemented version which accepts the same number and type of arguments. Our implemented versions have the same functionality as the original ones, and additional logic to generate the comparison progress information and inform the fuzzer.

The instrumentations help the fuzzer to keep the test inputs that can trigger program state changes, allowing the fuzzer to match the magic bytes comparison byte by byte. For instance, in a n-byte comparison, if we want to match all the whole n bytes correctly at once, the search space is in the complexity of 2^{8*n}. However, if we match the magic bytes byte by byte, the search space will be reduced to the complexity of $n * 2^8$, which is a magnitude reduction. Thus, the instrumentations contribute to solving the problem of how to perform mutations to match the magic bytes.

3.3 The Fuzzing Loop

The fuzzing loop takes the instrumented binary and starts fuzzing. The instrumentations (Section 3.2) can inform the fuzzer of whether a test input makes comparison/coverage progress or not. This information alone is still not sufficient to efficiently match the magic bytes as we do not know where the magic bytes locate in the test input. To know the location information of the magic bytes, one possible approach is to use taint analysis to extract the information

Algorithm 1: The Fuzzing Loop

```
1  S = ∅;
   // S is the test input queue
2  load S with the user provided test input(s);
3  while time budget reached or abort signal received do
4      s = NEXT(S);
       // s is the current input
5      if s is an intermediate step test input then
6          apply local exhaustive mutations (LEM);
           // n is the new test input
7          if LEM generates input n that triggers new program state then
8              append n to S;
9              if n improves comparison progress but not coverage then
10                 mark n as an intermediate step test input;
11                 keep the location of the mutated byte;
12             remove s from S;
13             continue;
14     apply normal mutations (NM);
       // n is the new test input
15     if NM generates input n that triggers new program state then
16         append n to S;
17         if n improves comparison progress but not coverage then
18             mark n as an intermediate step test input;
19             keep the location of the mutated byte;
20         if s is an intermediate step test input then
21             remove s from S;
```

of how the comparison operands are linked with the test input. For example, VUzzer [40] uses dynamic taint analysis to gather information of comparisons. However, though powerful and precise, taint analysis is not suitable for fast program execution [40]. Here, we propose an approach using feedback of the instrumentations to get the location information. The approach uses the heuristic that, if a byte of a test input is used in a comparison, then the bytes nearby may also be used in that comparison. In Steelix, after the fuzzer makes a mutation, if it is informed by the instrumentations that it makes progress in matching magic bytes, it will keep the new test input together with the position of the byte that it just mutated. When the fuzzer tries to mutate the new test input, it will exhaustively try all the possibilities of the two neighbor bytes according to different situations, i.e., *local exhaustive mutation*.

Algorithm 1 gives the procedure of the fuzzing loop, where the normal mutations and our local exhaustive mutation are applied adaptively. We will use Fig. 5 as an example to show how these mutations are guided by the coverage and comparison progress information. Assume that the magic string used for comparison is "MAZE" and the corresponding bytes in the initial input are "XXXX". The circles in Fig. 5 are the bytes that the fuzzer tries to mutate. First, the byte at offset 2 is mutated from "X" to "M", which can be easily achieved by normal mutation operators like bit-flipping or arithmetic plus/minus (Line 14). After this mutation, input 1 is generated. A coverage-based fuzzer will not be aware of this change and will discard input 1. However, the instrumentation used by Steelix will inform the fuzzer of hitting one byte of a magic bytes comparison (Line 15 and 17). Thus Steelix will keep input 1 as an *intermediate step* input, keep the location of the mutated byte, and discard input 0 (Line 17–21). When the fuzzer retrieves input 1 from the input queue, it will get this information: input 1 is an intermediate step input; and the last mutated location is at offset 2 (Line 5). Then, the fuzzer will try out all the 256 possibilities for the byte at offset 1 (Line 6). It will not get any feedback because the byte at offset 1 is just a dummy byte in this example. Thus the fuzzer will try with the byte at offset 3, and will be notified of making progress in matching bytes for comparisons when it generates input 2. The fuzzer will keep input 2, discard input 1, and continue fuzzing with input 2 (Line 7–13). When the fuzzer retrieves input 2 from the queue, it will receive two more pieces of information: input 2 is generated with the local exhaustive mutation; and input 2 is generated by mutating in the direction of increasing offsets (Line 5). Then, the fuzzer will only try out all the possibilities of the byte at offset 4. Like input 2, input 3 will be generated and used for generating input 4. When input 4 is generated and applied to execution, it will trigger a new execution path. The basic block instrumentation will inform the fuzzer that input 4 leads to new basic block coverage and it is no longer an intermediate step input. Thus the fuzzer will not apply the local exhaustive mutation on input 4.

From the example, we can see that with the comparison progress and location information, the fuzzer will generate a lot of inputs. However, not every one of them is of the same importance. In Steelix, the intermediate step inputs become useless after the input holding the final magic bytes is generated. Steelix will drop an intermediate step input from the queue once the fuzzer makes progress in solving magic byte comparisons based on that input (Line 12 and 21).

As discussed, the location information is based on the heuristic that the magic bytes used in a comparison are clustered in the input. However, it is possible that the magic bytes used in a comparison are from different parts of the input. In such a case, Steelix will not be informed of making any progress after applying the local exhaustive mutations. Steelix will keep the intermediate step input and apply the normal mutations on other bytes (Line 14), instead of removing that intermediate step input from the queue.

4 IMPLEMENTATION AND EVALUATION

We have implemented Steelix in Python, C and C++. Specifically, static analysis was implemented using IDAPython [20], and binary instrumentation was implemented using Dyninst [6]. We extended AFL 2.33b to be the fuzzer in Steelix. All the experimental results are available at our website [21].

4.1 Evaluation Setup

To evaluate the effectiveness of Steelix, we compared Steelix with several state-of-the-art fuzzers on a variety of programs.

Evaluation Datasets. We used two sets of widely-used benchmarks (i.e., LAVA-M [27] and DARPA CGC sample binaries [9]) and five real-life programs (i.e., `tiff2pdf`, `tiffcp`, `pngfix`, `gzip` and `tcpdump`) for our evaluation. The benchmark programs are known to have certain vulnerabilities, and hence form a ground-truth corpora for tool evaluation. The real-life programs are used to demonstrate the scalability and effectiveness of Steelix on large programs.

- **LAVA-M Dataset.** LAVA-M consists of 4 buggy version of Linux utilities, i.e., `base64`, `md5sum`, `uniq` and `who`. It was generated by automatically injecting hard-to-reach bugs into existing program source code [27], and was designed as a benchmark for evaluating the bug detection capability of fuzzers. The LAVA authors have demonstrated that a coverage-based fuzzer (FUZZER) and a SAT-based approach (SES) cannot find the injected bugs effectively [27]. Recent fuzzers (e.g., VUzzer [40]) all used this benchmark.
- **DARPA CGC Sample Binaries.** In 2016, The Defense Advanced Research Projects Agency (DARPA) [5] held a Cyber Grand Challenge (CGC), which was the first all-computer Capture the Flag tournament [9]. DARPA released the 141 binaries used in the qualification and final event of CGC and the 17 representative sample binaries. However, these binaries run under the DARPA Experimental Cyber Research Evaluation Environment (DECREE), while Steelix relies on Dyninst and it is hard to port Dyninst into DECREE. Although the team TrailofBits migrated these binaries into the Linux system [18], it is difficult to set up the fuzzing environment for some binaries due to some migration problems. Therefore, we only used the 17 representative sample binaries. Compared to LAVA-M, CGC sample binaries are smaller in size and contain fewer bugs per program, which are more suitable for detailed analysis of how Steelix helps bug detection.
- **Real-Life Programs.** We selected five real-life programs, i.e., `pngfix+libpng` [3], `tcpdump+libpcap` [1], `tiffcp+libtiff` [2], `tiff2pdf+libtiff` [2] and `gzip` [4], based on the following two criteria. First, each program is officially published together with the widely-used libraries. We also instrumented the libraries used by these programs to fuzz the program logic inside those libraries. Second, the programs/libraries represent different types of real-life programs. `pngfix`, `tiffcp` and `tcpdump` are about data parsing, while `tiff2pdf` and `gzip` are to perform calculations.

State-of-the-Art Tools. We compared Steelix with three most related state-of-the-art fuzzers: AFL-dyninst [11], VUzzer [40] and AFL-lafintel [17]. Note that Driller [44] is also closely related to Steelix, but its current release [19] was designed and built for only DECREE binaries. Thus we did not compare with Driller in the evaluation.

- AFL [8] only works on programs with source code provided. To enable AFL to fuzz program binaries effectively, researchers have proposed several extensions such as AFL-Qemu [12], AFLPIN [13] and AFL-dyninst [11]. Among them, AFL-dyninst is the closest one to Steelix since our implementation also relies on Dyninst. Thus we compared with AFL-dyninst on all the three datasets.
- VUzzer is a recently published fuzzer that also aims at solving the magic bytes comparison problem. The tool was not released when we conducted the experiments. Hence, we compared with VUzzer on the LAVA-M dataset using the data provided in their paper.
- AFL-lafintel, unlike AFL-dyninst, VUzzer and Steelix that work on binaries, requires the source code of the PUT. We decided to compare with AFL-lafintel as its divide-and-conquer approach is similar to our utilization of comparison progress, and used the LAVA-M dataset for the comparison across these four approaches.

Experimental Infrastructure. We ran all our experiments on a machine with 8 Intel(R) Xeon(R) CPU E5-1650 v3 cores and 8GB memory, running 32-bit Ubuntu 16.04 LTS system. Although AFL supports the *master-slave* fuzzing paradigm, for all the AFL-based fuzzers, including Steelix, AFL-dyninst and AFL-lafintel, we only used one master fuzzer instance for the experiments. This is not only to align with VUzzer, which does not support parallel fuzzing currently, but also to reduce the bias introduced by parallel fuzzing.

Table 1: Detected Bugs on LAVA-M Dataset

Program	Total Bugs	FUZZER	SES	VUzzer	AFL-lafintel	Steelix
base64	44	7	9	17	28	43 (26)
md5sum	57	2	0	1	0	28 (21)
uniq	28	7	0	27	24	7 (1)
who	2136	0	18	50	2	194 (77)
Total	2265	16	27	95	54	272 (125)

Research Questions. The experiments were designed to answer the following three research questions:

- **RQ1.** How good is the bug detection capability of Steelix?
- **RQ2.** How good is the code coverage of Steelix?
- **RQ3.** How is the overhead of Steelix on the fuzzing loop?

4.2 Results on LAVA-M Dataset (RQ1)

The authors of LAVA [27] evaluated a coverage-based fuzzer (FUZZER) and a SAT-based approach (SES) on the LAVA-M dataset for five hours, but did not reveal any detailed references of the two tools. Similarly, the authors of VUzzer [40] conducted the evaluation on the LAVA-M dataset for five hours, but the tool was not released when we conducted our experiments. Therefore, we also ran the experiments with AFL-dyninst, AFL-lafintel and Steelix for five hours, and just restated the results from the LAVA and VUzzer papers.

Table 1 reports the number of detected bugs with respect to these approaches. The first column reports the name of target programs. The second column shows the total number of bugs injected in each program. The third to fifth columns report the results from the LAVA and VUzzer paper. The last two columns gives the numbers of bugs detected by AFL-lafintel and Steelix. Note that AFL-dyninst did not find any bugs in five hours and we omitted its result. Each bug in LAVA-M has a unique ID, and the IDs of the bugs detected by Steelix in the experiments can be found at our website [21].

From Table 1, we can see that Steelix significantly outperformed SES and AFL-dyninst for all the programs; and it significantly outperformed FUZZER, VUzzer and AFL-lafintel for three out of the four programs. The reasons for the promising results are:

- Many bugs are injected in the execution paths protected by magic bytes comparisons. For example, the bug with ID 832 is triggered by mutating the bytes "las!" correctly at offset 56 of the test input. We also computed the number of bugs that were *directly found* by our local exhaustive mutation, and the results are reported in parentheses at the last column of Table 1. Our local exhaustive mutation can also generate test inputs that can lead to new execution paths but not crashes, and the crashing test inputs generated based on those test inputs are not included here. We can see that our local exhaustive mutation found 46% of the detected bugs. This indicates that our local exhaustive mutation guided by comparison progress is effective to pass magic bytes comparisons.
- Compared to VUzzer, Steelix enjoys a balance of execution speed and penetration power due to our light-weight nature, in contrast to the heavy-weight taint analysis used in VUzzer. Quantitatively, Steelix executed an average of 645× more inputs than VUzzer in five hours based on the data reported in their paper, which allows Steelix to run more test inputs. On the other hand, Steelix can handle `cmp`/`test` instructions with all operand types (memory reference, register and immediate values), while VUzzer can only handle `cmp`/`test` instructions with immediate value. In particular, around 80% of the comparisons involve immediate value in the four programs, which means VUzzer misses 20% of comparisons.
- Compared to AFL-lafintel, Steelix has the knowledge of the location of magic bytes in the test input, while AFL-lafintel does not know where to mutate the test input. Moreover, Steelix filters out uninteresting comparisons and marks comparison progress in a compact way, which is helpful for fuzzing large binaries, e.g., who. Instead, AFL-lafintel transforms the multi-byte comparisons into multiple nested one-byte comparisons, and adds many basic blocks to the PUT. For example, AFL-lafintel instrumented 99,866 locations in who, but the shared memory in AFL only has 65,536 entries, hindering AFL-lafintel from detecting coverage change of the PUT. In contrast, Steelix only instrumented 4,833 comparisons and 6,385 basic blocks for who, using around 25,000 entries in the shared memory. This explains why Steelix found many more bugs than AFL-lafintel on who and why we did not run AFL-lafintel on those large-size real-life programs in Section 4.4.

On the other hand, we can see that Steelix performed worse than VUzzer and AFL-lafintel on `uniq`, and found the same number of bugs as FUZZER. Comparing with VUzzer, the mutation operations used in Steelix are too fine-grained for the test input of `uniq`, which is a text file consisting of ASCII strings. The exhaustive mutation in Steelix is on the byte level and cannot quickly make large changes to the file. Instead, VUzzer uses crossover mutation in its genetic algorithm to exchange chunks of strings between test inputs, allowing it to quickly cover the code in `uniq`. Comparing with AFL-lafintel, our instrumentations, added by binary rewriting technique, is slower than AFL-lafintel's instrumentations added during the compilation. Particularly, in five hours, AFL-lafintel conducted 32.6 million test executions, while AFL-dyninst and Steelix only performed 5.3 million and 4.5 million test executions respectively. Although relying on the source code, the speed advantage allows AFL-lafintel to find more bugs in `uniq` within five hours.

From the analysis of the LAVA-M experimental results, we can positively answer **RQ1** that Steelix significantly outperformed the state-of-the-art fuzzers in terms of bug detection capability.

4.3 Results on CGC Sample Binaries (RQ1)

For the 17 representative CGC sample binaries, we filtered out some of the binaries for the following reasons:

- Some binaries are duplicates of others. For example, CADET00003 is a duplicate of CADET00001 with only some minor changes. Thus, experiments on these two binaries will produce the same result.
- Some binaries have interaction with others. For example, LUNGE-00005 involves inter-process communication of six binaries. Such binaries are not applicable to fuzzing, and thus are removed.
- It is hard to generate valid initial test inputs for some binaries, as some inputs are interactively generated by including part of the program output and some inputs are in binary format. For example, KPRCA00003 is an image compressor which accepts custom-defined images as inputs.

After the filtering, we used eight CGC sample binaries in this experiment to analyze how Steelix helps find bugs, comparing with existing coverage-based fuzzers. We ran Steelix and AFL-dyninst

Table 2: Detected Bugs on CGC Sample Binaries

Program	AFL-dyninst	Steelix
CADET00001	✓	✓
EAGLE00005	✓	✓
NRFIN00010	✓	✓
YAN0100001	✓	✓
YAN0100003	✓	✓
KPRCA00001	✗	✓
KPRCA00015	✓	✓
YAN0100002	✗	✗

```
ïż£static int page_root64(char *input) {
  char buf[256];
  const char *mode = variable_get("mode");
  ...
  if (strcmp(mode, "encode") == 0)
    // vulnerable code
  else if (strcmp(mode, "decode") == 0)
    ...
  ...
}
```

Figure 6: The page_root64 Function in KPRCA00001

on them for three hours. Table 2 shows the overall results of the experiment. Specifically, AFL-dyninst found the bugs in six out of the eight binaries, and Steelix found the bugs in seven binaries.

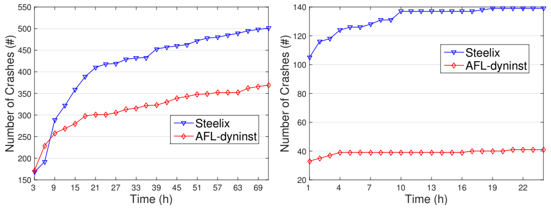
We analyzed these binaries in detail and had some insightful findings. For YAN0100002, both AFL-dyninst and Steelix failed to find any bugs in three hours. This binary is a tennis ball motion calculator, and the bug in this binary is modeled against "The Patroit Missile Failure" [7]. The bug is triggered only when a test input can cause the program to perform millions of floating-point calculations. Thus, the bug is not related to magic bytes comparison, and in such cases Steelix cannot improve the bug detection capability.

For KPRCA00001, the binary is a root64 encoder/decoder. The bug is a write-out-of-bound in the root64 encoding function. If we provide an input that can trigger the encoding function, both AFL-dyninst and Steelix can find the bug with around 8 minutes. However, if the initial input cannot trigger the encoding function, AFL-dyninst can hardly find the bug in 3 hours, while Steelix can still trigger the crash with around 10 minutes. This is because in the `page_root64` function of this binary (as shown in Fig. 6), the variable `mode` extracted from the input is compared with string "encode" and "decode"; and Steelix can capture such magic bytes comparison, and rely on local exhaustive mutation to trigger the crash.

> From the analysis of the CGC experiment results, we can positively answer **RQ1** that given good initial inputs, both AFL-dyninst and Steelix can efficiently find bugs; however, when the initial input is not so desirable, Steelix has a better potential to find bugs thanks to its penetration power.

4.4 Results on Real-Life Programs (RQ1 & RQ2)

We compared Steelix with AFL-dyninst on the real-life programs in terms of crashes, bugs and code coverage. `pngfix`, `tcpdump` and `gzip` were fuzzed for 24 hours, while `tiffcp` and `tiff2pdf` were fuzzed for 72 hours as they are much larger in size.



(a) tiffcp **(b) gzip**

Figure 7: Number of Unique Crashes Detected over Time

Table 3: New Bugs Detected in Real-Life Programs

Program	Bug Type	AFL-dyninst	Steelix
tiffcp	Out-of-Bounds Read (CVE-2017-5225)	✓	✓
tiffcp	Out-of-Bounds Write	✓	✓
tiff2pdf	Out-of-Bounds Write	✗	✓
gzip	Out-of-Bounds Write	✗	✓
gzip	Out-of-Bounds Write	✗	✓
gzip	Out-of-Bounds Read	✓	✓
gzip	Double-Free	✓	✓
gzip	Null-Pointer Dereference	✓	✓
gzip	Null-Pointer Dereference	✓	✓
gzip	Null-Pointer Dereference	✓	✓

4.4.1 Crash and Bug Analysis. Steelix found 500, 139 and 1 unique crashes in `tiffcp`, `gzip` and `tiff2pdf` respectively, and AFL-dyninst found 367 and 41 unique crashes in `tiffcp` and `gzip` respectively. For `pngfix` and `tcpdump`, Steelix and AFL-dyninst did not trigger any crashes. A crash is considered as unique if at least one of its associated execution paths is not seen in previously-recorded crashes, following the definition in AFL [8]. Further, Fig. 7 shows the number of unique crashes over time. We can see that Steelix can keep finding more unique crashes than AFL-dyninst.

After analyzing these crashes, we found ten previously unknown bugs and one of them has been accepted as a CVE, which are listed in Table 3 together with their bug types. Specifically, we found two out-of-bounds read bugs, four out-of-bounds write bugs, one double-free bug, and three null-pointer dereference bugs. More details about these bugs can be found at our website [21]. However, AFL-dyninst failed to find three of them, which were protected by magic bytes comparisons. Instead, Steelix successfully found them with the help of our local exhaustive mutations.

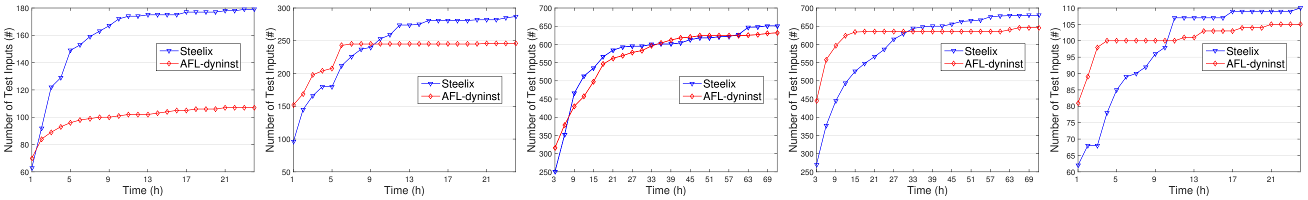
4.4.2 Coverage Analysis. Table 4 reports the number of lines, functions and branches covered by AFL-dyninst and Steelix. Overall, the improvement of Steelix over AFL-dyninst on programs that perform parsing (`pngfix`, `tcpdump` and `tiffcp`) is respectively 12.7%, 9.7% and 11.4% in terms of lines of code, functions and branches, which is much larger than the improvement on programs that perform calculations (`tiff2pdf` and `gzip`). This is because the parser programs involve more magic bytes comparisons in their logic and Steelix is designed to tackle this problem.

More specifically, the coverage improvement on `pngfix` is the largest, while the coverage improvement on `gzip` is the smallest. For `pngfix`, the inputs are png files that contain four-byte chunk-type strings which describe the type of chunk-data and are used to control the parsing process. Matching the chunk types correctly will greatly increase the code coverage. For example, Steelix covered the codes dealing with the `zTXt`, `iCCP` and `iTXt` chunks, while AFL-dyninst failed to generate inputs with these chunks due to the lack of an effective way to match magic byte comparisons.

For `gzip`, its main logic is about the compression and decompression of data. Unlike png files, the local header bytes in a zip file

Table 4: Line, Function and Branch Covered by Steelix and AFL-dyninst

Program	AFL-dyninst			Steelix		
	Line (#)	Function (#)	Branch (#)	Line (#)	Function (#)	Branch (#)
pngfix + libpng	2098	136	1163	2540 (+21.1%)	155 (+14.0%)	1450 (+24.7%)
tcpdump + libpcap	3142	194	1775	3467 (+10.3%)	203 (+4.6%)	1994 (+12.3%)
tiffcp + libtiff	5528	296	3706	5889 (+6.5%)	327 (+10.5%)	3913 (+5.6%)
tiff2pdf + libtiff	6718	325	3950	6931 (+3.2%)	331 (+1.9%)	4072 (+3.1%)
gzip	842	41	620	855 (+1.5%)	41 (0%)	627 (+1.1)



(a) pngfix (b) tcpdump (c) tiffcp (d) tiff2pdf (e) gzip

Figure 8: The Number of Generated Test Inputs that Cover New Basic Blocks over Time

are used for calculation but not for comparison. This limits the improvement of Steelix over AFL-dyninst. Nevertheless, `gzip` is the smallest amongst the five real-life programs, and both AFL-dyninst and Steelix can cover most of the code in `gzip` and the improvement of Steelix becomes less significant.

Moreover, we also computed the number of generated inputs that covered new basic blocks over time. Note that we did not include the generated inputs that made intermediate comparison progress. The results are shown in Fig. 8. We can see that Steelix converges slower than AFL-dyninst because of the overhead (see detailed discussion in Section 4.5) caused by our extra instrumentations and extra logic in the fuzzing loop. However, we can also see that Steelix can keep generating new interesting inputs and thus have more interesting inputs in the long run. This is because Steelix can break through magic bytes comparisons and penetrate deeper than AFL-dyninst.

From the analysis of the real-life programs, we can positively answer **RQ1** and **RQ2** that Steelix outperformed the state-of-the-art coverage-based fuzzer in terms of both bug detection capability and code coverage thanks to its penetration power.

4.5 Overhead Evaluation (RQ3)

To evaluate the execution speed overhead of Steelix, we first reported the instrumentation statistics of Steelix, and compared Steelix to AFL-dyninst in terms of the number of executions of test inputs.

Recall that AFL-dyninst instruments every basic block and Steelix instruments basic blocks and comparisons. Table 5 reports the number of basic blocks (column *BB*), the number of comparisons filtered out by the two rules in Section 3.1.2 (i.e., one-byte comparison (column *One-Byte CMP*) and function return value comparison (column *FRV CMP*)), and the number of instrumented comparisons (column *Instrumented CMP*). We can see that Steelix added an average of around 40% more instrumentations. Besides, on average, 14.2% and 3.5% are one-byte comparisons and function return value comparisons respectively, which were filtered out by Steelix.

Moreover, Fig. 9 shows the number of test input executions on each real-life program. We can see that Steelix executed slightly slower than AFL-dyninst for all programs except for `pngfix`. For `pngfix`, Steelix had more input executions than AFL-dyninst. This is because the execution speed is also affected by the quality of the inputs. If the fuzzer keeps executing the inputs that exercise slow execution paths, the execution speed will be heavily affected. Quantitatively, the 40% more instrumentations introduced a 11.5% smaller of test input executions, but covered more code and found more bugs. This indicates that the overhead is small and acceptable.

From the analysis of Table 5 and Fig. 9, we can positively answer **RQ3** that Steelix introduced a small and acceptable overhead with respect to the execution speed.

4.6 Discussion

From our evaluations, we can conclude that Steelix achieves better bug detection and code coverage than the current state-of-the-art fuzzers at a reasonable cost. Nevertheless, we want to highlight that Steelix cannot handle all magic bytes comparisons effectively. First, the comparisons of function return values, filtered out by Steelix, are not handled by Steelix. Second, the heuristic that the magic bytes used in a comparison are clustered in the input might not always hold, and thus our instrumentation might be less effective to break through magic bytes comparisons. In such cases, symbolic execution or taint analysis can be helpful to infer how the input data is linked with the program instructions. Therefore, Steelix cannot replace symbolic execution or taint analysis in every case, but it can relieve some burden from them because not every comparison requires symbolic execution or taint analysis to solve.

5 RELATED WORK

In this section, we focus our discussion on mutation-based fuzzing as Steelix is also mutation-based. Note that a number of advances have also been made to improve the effectiveness and efficiency of generation-based fuzzing [10, 14, 26, 30, 35, 39, 42, 45, 46, 50].

5.1 Symbolic-Based Approaches

Symbolic/concolic execution are commonly used to perform or help fuzzing. SAGE [31, 32] leverages symbolic analysis on execution

Table 5: The Number of Instrumented Comparisons

Program	BB (#)	One-Byte CMP (#)	FRV CMP (#)	Instrumented CMP (#)
pngfix + libpng	8166	963 (23.1%)	115 (2.8%)	3090 (74.1%)
tcpdump + libpcap	7666	433 (12.1%)	142 (4.0%)	3001 (83.9%)
tiffcp + libtiff	8814	638 (14.7%)	112 (2.6%)	3586 (82.7%)
tiff2pdf + libtiff	10626	687 (13.6%)	169 (3.3%)	4206 (83.1%)
gzip	1542	70 (7.7%)	43 (4.7%)	801 (87.6%)

traces and relies on constraint solving to generate tests. By contrast, several works use symbolic analysis to help fuzzing. SYMFUZZ [25] applies symbolic analysis on a program-input pair to detect dependencies among the bit positions of the input. The dependency information is used to compute the optimal mutation ratio. Babić et al. [23] uses symbolic execution to generate test inputs and direct its exploration to trigger the target potential vulnerabilities.

Driller [44], one of the closest work to Steelix, combines fuzzing and concolic execution. Specifically, whenever the fuzzer gets stuck at some magic bytes comparison, it uses concolic execution to generate a test input that can pass that comparison. Differently, Steelix attempts to pass magic bytes comparisons with the mutation-based fuzzer itself by the guidance provided by our light-weight program instrumentation. Steelix can relieve some burden from symbolic/-concolic execution, but cannot replace symbolic/concolic execution as Steelix is not suitable for some magic bytes comparisons.

5.2 Taint-Based Approaches

Taint analysis can extract the relations between the data in the test input and the logic of the program. Researchers have proposed to use taint analysis to provide guidance for fuzzing. BuzzFuzz [29] and VUzzer [40] use dynamic taint analysis to locate the interesting bytes for the fuzzer to mutate. Besides, several works attempt to combine taint analysis and symbolic execution to guide the fuzzing. TaintScope [48] performs checksum-aware fuzzing. It identifies the bytes that are used in security-sensitive operations by taint tracking, and generates inputs that are more likely to trigger potential vulnerabilities using combined concrete and symbolic execution techniques. Dowser [33] targets buffer overflow and underflow vulnerabilities. It uses taint analysis to determine the input bytes that influence the array index, and then uses symbolic execution to generate the test inputs that trigger the vulnerability. Similarly, BORG [38] targets buffer over-read bugs. It works by first using taint analysis to select buffer accesses and then guiding symbolic execution towards those accesses to detect over-read.

Taint analysis can help to precisely locate the magic bytes in the test input, but may have high performance overhead and slow down the execution speed of fuzzers due to its heavy-weight nature. Differently, Steelix uses light-weight instrumentation to help track such taint information, which is not very precise but can provide sufficient guidance to the fuzzers efficiently.

5.3 Fuzzing Boosting

Several boosting techniques have been proposed to improve the efficiency of mutation-based fuzzing. Rebert et al. [41] and Woo et al. [49] empirically studied how the seed selection algorithms and fuzzing scheduling strategies can help maximize the bug detection capability of a fuzzer. AFLFast [24] speeds up AFL by focusing the fuzzing efforts on low-frequency paths, which allows the fuzzer to explore more paths with the same amount of time. Skyfire [47] leverages the knowledge in existing samples to generate well-distributed seed inputs for fuzzing programs that process highly-structured inputs. Kargén and Shahmehri [36] take a different perspective to perform the fuzzing. They perform mutations on the machine code of the generating programs instead of directly on a well-formed input. In this way, they can use the information about the input format encoded in the generating program to produce high-coverage test inputs. Steelix is orthogonal to these boosting techniques. We plan to combine them with Steelix and evaluate whether the effectiveness or efficiency of fuzzing can be further improved.

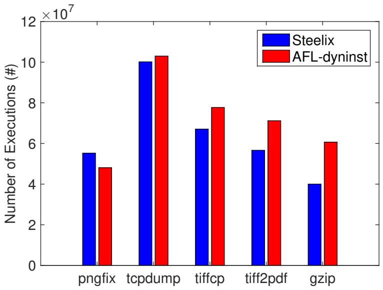


Figure 9: The Number of Executions on Real-Life Programs

Besides, AFL-lafintel [17] applies program transformation at LLVM IR level to convert a magic bytes comparison into multiple nested one-byte comparisons. Such transformations can help the fuzzer to keep the test inputs that make progress in comparisons, which is similar to Steelix. However, such transformations can also prevent the fuzzer from discriminating and discarding such *intermediate step* test inputs, which makes the fuzzer spammed with less interesting test inputs. Instead, Steelix can discard such intermediate step test inputs. Besides, AFL-lafintel works at the source code level, but Steelix directly works at the binary level. AFL-lafintel also fails to know where the magic bytes are located in the test input, which limits its effectiveness.

6 CONCLUSION

In this paper, we have proposed a program-state based binary fuzzing approach, named Steelix. The program state information kept by Steelix contains not only coverage information but also the comparison progress information, which are gathered by our light-weight static analysis and binary instrumentation. With the guidance of the comparison progress, Steelix can penetrate magic bytes comparisons more effectively and efficiently than traditional coverage-based fuzzers while keeping the execution overhead at a low level. We have implemented Steelix and evaluated it on various programs. The results demonstrate that Steelix can effectively improve fuzzing with respect to bug detection capability and code coverage.

ACKNOWLEDGMENTS

This research is supported by the National Research Foundation, Singapore under its National Cybersecurity R&D Program (Award No. NRF2014NCR-NCR001-30), the project M4081588.020.500000, the National Natural Science Foundation of China (Grant No. 61370079) and the NTU Start-Up grant (M4081190).

REFERENCES

[1] 1999. Tcpdump & Libpcap. (1999). http://www.tcpdump.org.

[2] 2001. Libtiff. (2001). http://www.libtiff.org.

[3] 2002. Libpng. (2002). http://www.libpng.org.

[4] 2003. Gzip. (2003). http://www.gzip.org.

[5] 2005. Defense Advanced Research Projects Agency. (2005). http://www.darpa.mil/.

[6] 2005. Dyninst API. (2005). http://www.dyninst.org/dyninst.

[7] 2006. The Patroit Missile Failure. (2006). https://www.ima.umn.edu/~arnold/disasters/patriot.html.

[8] 2014. American fuzzy lop. (2014). http://lcamtuf.coredump.cx/afl/.

[9] 2014. Cyber Grand Challenge. (2014). http://archive.darpa.mil/cybergrandchallenge/about.html.

[10] 2014. Spike fuzzer platform. (2014). http://www.immunitysec.com/.

[11] 2015. AFL-dyninst. (2015). https://github.com/vrtadmin/moflow/tree/master/afl-dyninst.

[12] 2015. AFL-QEMU. (2015). http://lcamtuf.coredump.cx/afl/technical_details.txt.

[13] 2015. AFLPIN. (2015). https://github.com/mothran/aflpin.

[14] 2015. Peach fuzzer platform. (2015). http://www.peachfuzzer.com/products/peach-platform/.

[15] 2015. Sdl Process: Verification. (2015). https://www.microsoft.com/en-us/sdl/process/verification.aspx.

[16] 2016. The bug-o-rama trophy case of AFL. (2016). http://lcamtuf.coredump.cx/afl/#bugs.

[17] 2016. Circumventing fuzzing roadblocks with compiler transformations. (2016). https://lafintel.wordpress.com/2016/08/15/circumventing-fuzzing-roadblocks-with-compiler-transformations/.

[18] 2016. DARPA Challenge Binaries on Linux and OS X. (2016). https://github.com/trailofbits/cb-multios/.

[19] 2016. Driller Source Code. (2016). https://github.com/shellphish/driller.

[20] 2016. IDAPython. (2016). https://www.hex-rays.com/products/ida/support/idapython_docs/.

[21] 2017. Steelix. (2017). https://sites.google.com/site/steelix2017/.

[22] Brad Arkin. 2009. Adobe Reader and Acrobat Security Initiative. (2009). http://blogs.adobe.com/security/2009/05/adobe_reader_and_acrobat_secur.html.

[23] Domagoj Babić, Lorenzo Martignoni, Stephen McCamant, and Dawn Song. 2011. Statically-directed Dynamic Automated Test Generation. In *ISSTA*. 12–22.

[24] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2016. Coverage-based Greybox Fuzzing as Markov Chain. In *CCS*. 1032–1043.

[25] Sang Kil Cha, Maverick Woo, and David Brumley. 2015. Program-adaptive mutational fuzzing. In *SP*. 725–741.

[26] Kyle Dewey, Jared Roesch, and Ben Hardekopf. 2014. Language Fuzzing Using Constraint Logic Programming. In *ASE*. 725–730.

[27] Brendan Dolan-Gavitt, Patrick Hulin, Engin Kirda, Tim Leek, Andrea Mambretti, Wil Robertson, Frederick Ulrich, and Ryan Whelan. 2016. LAVA: Large-Scale Automated Vulnerability Addition. In *SP*. 110–121.

[28] Chris Evans, Matt Moore, and Tavis Ormandy. 2011. Google online security blog – Fuzzing at scale. (2011). https://security.googleblog.com/2011/08/fuzzing-at-scale.html.

[29] Vijay Ganesh, Tim Leek, and Martin Rinard. 2009. Taint-based directed whitebox fuzzing. In *ICSE*. 474–484.

[30] Patrice Godefroid, Adam Kiezun, and Michael Y Levin. 2008. Grammar-based whitebox fuzzing. In *PLDI*. 206–215.

[31] Patrice Godefroid, Michael Y. Levin, and David Molnar. 2008. Automated white-box fuzz testing. In *NDSS*.

[32] Patrice Godefroid, Michael Y. Levin, and David Molnar. 2012. SAGE: Whitebox Fuzzing for Security Testing. *Commun. ACM* 55, 3 (2012), 40–44.

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

[34] Niranjan Hasabnis and R. Sekar. 2016. Lifting Assembly to Intermediate Representation: A Novel Approach Leveraging Compilers. In *ASPLOS*. 311–324.

[35] Christian Holler, Kim Herzig, and Andreas Zeller. 2012. Fuzzing with code fragments. In *USENIX Security*. 445–458.

[36] Ulf Kargén and Nahid Shahmehri. 2015. Turning programs against each other: high coverage fuzz-testing using binary-code mutation and dynamic slicing. In *FSE*. 782–792.

[37] Barton P. Miller, Louis Fredriksen, and Bryan So. 1990. An Empirical Study of the Reliability of UNIX Utilities. *Commun. ACM* 33, 12 (1990), 32–44.

[38] Matthias Neugschwandtner, Paolo Milani Comparetti, Istvan Haller, and Herbert Bos. 2015. The BORG: Nanoprobing Binaries for Buffer Overreads. In *CODASPY*. 87–97.

[39] Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. 2016. Model-based whitebox fuzzing for program binaries. In *ASE*. 543–553.

[40] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. 2017. VUzzer: Application-aware Evolutionary Fuzzing. In *NDSS*.

[41] Alexandre Rebert, Sang Kil Cha, Thanassis Avgerinos, Jonathan Foote, David Warren, Gustavo Grieco, and David Brumley. 2014. Optimizing seed selection for fuzzing. In *USENIX Security*. 861–875.

[42] Jesse Ruderman. 2007. Introducing jsfunfuzz. (2007). http://www.squarefree.com/2007/08/02/introducing-jsfunfuzz

[43] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2016. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *S&P*. 138–157.

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

[45] Spandan Veggalam, Sanjay Rawat, Istvan Haller, and Herbert Bos. 2016. IFuzzer: An Evolutionary Interpreter Fuzzer Using Genetic Programming. In *ESORICS*. 581–601.

[46] Joachim Viide, Aki Helin, Marko Laakso, Pekka Pietikäinen, Mika Seppänen, Kimmo Halunen, Rauli Puuperä, and Juha Röning. 2008. Experiences with Model Inference Assisted Fuzzing. In *WOOT*. 2:1–2:6.

[47] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2017. Skyfire: Data-Driven Seed Generation for Fuzzing. In *SP*. 579–594.

[48] Tielei Wang, Tao Wei, Guofei Gu, and Wei Zou. 2010. TaintScope: A Checksum-Aware Directed Fuzzing Tool for Automatic Software Vulnerability Detection. In *SP*. 497–512.

[49] Maverick Woo, Sang Kil Cha, Samantha Gottlieb, and David Brumley. 2013. Scheduling black-box mutational fuzzing. In *CCS*. 511–522.

[50] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and Understanding Bugs in C Compilers. In *PLDI*. 283–294.