SLF: Fuzzing without Valid Seed Inputs

Wei You[1], Xuwei Liu[2], Shiqing Ma[1], David Perry[1], Xiangyu Zhang[1], Bin Liang[3]

[1]Department of Computer Science, Purdue University, Indiana, USA

[2]School of Computer Science and Technology, Zhejiang University, Zhejiang, China

[3]School of Information, Renmin University of China, Beijing, China

Email: {you58, ma229, perry74, xyzhang}@purdue.edu, xuweiliu@zju.edu.cn, liangb@ruc.edu.cn

***Abstract*—Fuzzing is an important technique to detect software bugs and vulnerabilities. It works by mutating a small set of seed inputs to generate a large number of new inputs. Fuzzers' performance often substantially degrades when valid seed inputs are not available. Although existing techniques such as symbolic execution can generate seed inputs from scratch, they have various limitations hindering their applications in real-world complex software. In this paper, we propose a novel fuzzing technique that features the capability of generating valid seed inputs. It piggy-backs on AFL to identify input validity checks and the input fields that have impact on such checks. It further classifies these checks according to their relations to the input. Such classes include arithmetic relation, object offset, data structure length and so on. A multi-goal search algorithm is developed to apply class-specific mutations in order to satisfy inter-dependent checks all together. We evaluate our technique on 20 popular benchmark programs collected from other fuzzing projects and the Google fuzzer test suite, and compare it with existing fuzzers AFL and AFLFast, symbolic execution engines KLEE and S2E, and a hybrid tool Driller that combines fuzzing with symbolic execution. The results show that our technique is highly effective and efficient, out-performing the other tools.**

I. Introduction

Fuzzing is a commonly used technique to discover software bugs and vulnerabilities. It derives a large number of new inputs from some initial inputs called *seed inputs*, using mutations guided by various heuristics with a certain degree of randomness. It usually does not rely on complex program analysis or even source code, and hence features applicability. Many existing CVE reports were generated by fuzzing techniques [1], [2].

Valid seed inputs play a critical role in fuzzing. However, there are situations that seed inputs are not available or the available seed inputs may not cover important input formats. The former situation occurs for software that does not come with inputs or input specification (e.g., those downloaded from the Internet, third-party libraries, or even malicious code). The latter situation arises for software that supports many formats (including even non-standard/undocumented formats). Some software implementations may not fully respect documented specification, leading to implementation-specific input formats. They are unlikely to be represented in seed inputs. We will show in Section IV-B that a mutated `TIFF` format is uniquely supported by Exiv2 but completely undocumented. In fact, some CVEs (e.g., CVE-2010-3269 [3] and CVE-2016-4655 [4]) were discovered with inputs of undocumented format.

Hence, there is a strong need to fuzz software without requiring valid seed inputs. While most fuzzing techniques can operate without a valid seed input, they have to compose valid inputs from scratch through a more-or-less random search procedure such as random growth of the input length and random bit flipping, which have limited capabilities in generating valid inputs due to the large search space.

There are learning based techniques that aim to derive input specification from a large number of training inputs [5], [6]. They require the training set to have comprehensive coverage of input specification. Such an assumption is difficult to meet in practice. Moreover, as we discussed earlier, many inputs even have non-standard, undocumented, implementation specific formats. There are reverse-engineering techniques such as TIE [7] or REWARD [8] that can derive the grammatical structure of an input by monitoring program execution on the input. They hardly generate the entire input specification but rather the syntactical structure of individual concrete inputs.

Symbolic execution is a highly effective testing technique that can generate inputs from scratch [9]–[13]. Despite its effectiveness in unit testing, applying symbolic execution to whole system testing has a number of practical hindrances. For example, a lot of libraries need to be manually modeled; inputs with nontrivial formats often involve operations that are difficult for symbolic execution engines such as those in checksum computation; and many input validity checks require reasoning about the offsets, counts, and length of input fields, which are difficult for most symbolic execution engines. There are proposals to use gradients to guide fuzzing to provide the abilities of resolving path conditions in a non-random fashion [14]; and to combine symbolic execution and fuzzing [15]. However, they do not focus on solving the problem of generating valid seed inputs and hence still have the aforementioned limitations in our target setting.

In this paper, we develop a novel fuzzing technique SLF (stands for *Seedless Fuzzer*) that features valid seed input generation. It is fuzzing based and hence has the benefits of applicability. It does not require source code, nor does it rely on complex program analysis. Its operation is largely piggy-backing on regular fuzzing. Specifically, it starts with a very short random input (e.g., 4 bytes), which quickly fails some validity check. SLF then performs sophisticated input mutation to get through these validity checks until a valid seed input is generated, which is further subject to regular fuzzing. To get through validity checks, it first groups inputs into

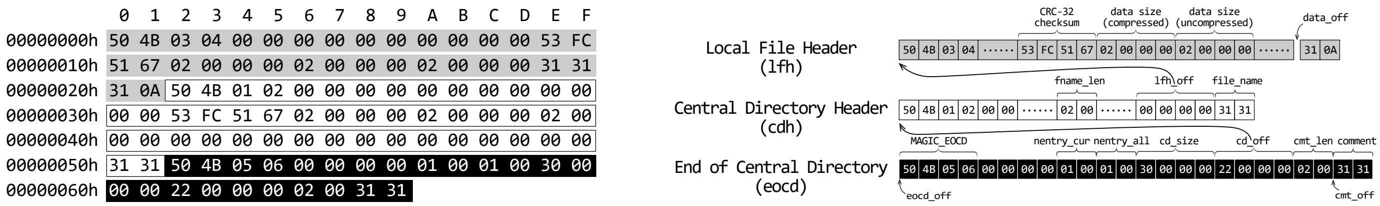


Fig. 1: A valid ZIP archive file.

fields by observing how consecutive bytes influence individual checks. Such information can be extracted from the underlying fuzzing infrastructure, which is AFL [16] in our case. It then classifies the checks based on their relations with the input, by observing the state differences at the checks caused by a pre-defined set of mutations. For example, some checks concern the values of individual input fields while others test the offset/count of specific fields. Such category information can be used as guidance in applying the corresponding mutations. In practice, multiple validity checks are often inter-dependent by predicating on a same variable or multiple variables derived from the same input fields. As part of SLF, we develop a search algorithm that can mutate input in a way to satisfy the inter-dependent checks.

We evaluate SLF on 20 real-world programs collected from the existing fuzzing projects [17], [18] and the standard Google fuzzer test suite [19] and compare it with existing fuzzers (AFL [16] and AFLFast [20]), symbolic execution engines (KLEE [9] and S2E [13]), and a combination of fuzzing and symbolic execution (Driller [15]). Our results show that SLF is highly effective and efficient. It can handle many more cases than the other techniques, and generate 7.3 times more valid seed inputs and cover 3.5 times more paths on average.

We make the following contributions.

- We propose a novel fuzzing technique that can effectively generate valid seed inputs. The technique is piggy-backing on AFL and does not require any complex program analysis or even source code.
- We develop various supporting techniques, such as grouping input bytes to fields; detecting inter-dependent input validity checks; and a search algorithm to satisfy these checks together.
- We propose a classification of input validity checks based on their relations with input elements. We also develop the corresponding mutation strategies.

II. Motivation

We use LibZip [21] to explain the limitations of existing techniques (e.g., mutation-based fuzzing and symbolic execution) and motivate the idea of SLF. LibZip is an open-source library that reads, creates and modifies `ZIP` archives. It is widely integrated in commodity software, such as PDF readers and SQL databases, for data compression and decompression.

ZIP File Format. ZIP is one of the most popular compressed file formats [22]. A ZIP archive must contain an "end of central directory" (`eocd` for short) structure located at the end of the file (e.g., bytes in black at the index 0x52-69 in Figure 1). Each file stored in a ZIP archive is described by a "local file header" (`lfh`) structure, which records the basic information about the file, such as data size and CRC-32 checksum (e.g., the bytes in gray at index 0x00-1F). The file data is placed after the local file header (index 0x20-21). For ease of indexing, each file is accompanied with a record of "central directory header" (`cdh`), which indicates the offset of the corresponding `lfh` and is located in a central place of the archive together with the `cdh` records of other files (e.g., plain bytes at 0x22-51 pointing to the aforementioned `lfh`). The `eocd` holds a pointer pointing to the `cdh` section. Note that the valid ZIP file in Figure 1 is generated by our tool from scratch.

Assume that we are using LibZip to read the content of a specified file in a given ZIP archive. LibZip first identifies the `eocd` structure, from which it can locate the `cdh` records. It then iterates through the individual records until the one for the specified file is found. Finally, the corresponding `lfh` structure is accessed and the file data is extracted. During the process, LibZip performs multiple checks on different fields of the input file to ensure validity of the archive. If any check fails, the process is terminated. Figure 2 presents some critical checks. Check ⓐ ensures the input file has enough length to hold an `eocd` structure. Check ⓑ searches for the magic number of `eocd` structure in the input file. If found, its offset is recorded in the *eocd_off* variable. After that, LibZip examines whether the number of `cdh` records on the current disk equals to the the total number of `cdh` records (check ⓒ) and whether it is larger than the index of the specified file (check ⓓ) [1]. Subsequently, the size (*cd_size*) and offset (*cd_off*) of the `cdh` section are extracted from the input file at the offset *eocd_off*+12 and *eocd_off*+16, respectively (e.g., indices 0x5E-61 and 0x62-65 in Figure 1). Then, LibZip checks whether the `cdh` section overlaps with the `eocd` structure (check ⓔ) and whether its size is larger than the default (minimal) size of a `cdh` record (check ⓕ). Next, LibZip gets the length of the comment string (*cmt_len*) and checks whether it equals to the length of the remaining data (check ⓖ). Finally, the integrity of the archive is inspected by comparing the computed CRC-32 checksum with the one stored in the archive (check ⓗ).

Mutation-based Fuzzing. Mutation-based fuzzing [16], [17], [20] or fuzzing in short generates inputs by modifying valid seed inputs. Although many fuzzing techniques can operate without a valid seed input, their effectiveness often substan-

[1]Note that LibZip does not support cross-disk zip archive and check ⓒ ensures the condition is satisfied. *SPECIFIED_INDEX* in check ⓓ is a command line parameter that indicates the index of the file to be accessed.

```
01 int file_size = size_of_file();
02 char *buf = read_from_file();
03 int min_size = is_zip64(buf)?SIZE_EOCD64:SIZE_EOCD;
04 if (file_size < min_size) error();                  ⓐ
05 int eocd_off = 0;
06 while (memcmp(buf + eocd_off, MAGIC_EOCD, 4) != 0)
07   eocd_off++;                                        ⓑ
08 if (eocd_off > file_size) error();
09 int nentry_cur = read_buffer16(buff, eocd_off + 8);
10 int nentry_all = read_buffer16(buff, eocd_off + 10);
11 if (nentry_cur != nentry_all) error();              ⓒ
12 if (nentry_cur <= SPECIFIED_INDEX) error();         ⓓ
13 int cd_size = read_buffer32(buf, eocd_off + 12);
14 int cd_off = read_buffer32(buf, eocd_off + 16);
15 if (cd_size + cd_off > eocd_off) error();           ⓔ
16 if (cd_size < SIZE_CDH) error();                    ⓕ
17 int cmt_len = read_buffer16(buf, eocd_off + 20);
18 int tail_len = buffer_left(buf);
19 if (cmt_len != tail_len) error();                   ⓖ
20 int crc_cmp = compute_crc(buf, data_off, data_size);
21 int crc_str = read_buffer32(buf, lfh_off + 14);
22 if (crc_cmp != crc_str) error();                    ⓗ
```

Fig. 2: Critical validation checks of LibZip.

tially degrades. While some checks in Figure 2 can be satisfied by random mutation with dictionary (e.g., checks ⓐ and ⓑ), other checks are very difficult to satisfy. For example, check ⓖ constrains the file size with a value extracted from the input file. Randomly mutating the input bytes or extending the file size are unlikely to satisfy the check. In fact, both AFL [16] and AFLFast [20] would get stuck at similar checks.

To overcome some of these difficulties, Angora [14] enhances fuzzing with gradient guided mutation, which considers a path condition as a black-box function on inputs and searches for input valuations that satisfy the condition. It uses taint analysis to identify the input bytes that affect a predicate, observes the changes on the predicate after mutating the identified bytes to derive the direction for changes (i.e., if/how-significant individual input changes impact the comparison at the predicate), and performs further mutations accordingly. Although achieving substantial improvement over random fuzzing, it has inherent limitations in satisfying multiple path conditions involving the same variable. For example, when computing the gradient for satisfying check ⓓ, Angora only tries to change the value of *nentry_cur* without keeping it equal to *nentry_all*, which is required by check ⓒ. In such cases, Angora could hardly find the direction of input changes.

Symbolic Execution. Symbolic execution [9]–[13] uses symbolic variables to denote individual input elements and then constructs symbolic constraints (i.e., formulas involving symbolic variables) to represent the conditions that need to satisfy in order to explore various program paths. Concrete inputs are derived by solving these constraints. While symbolic execution can generate seed inputs from scratch and handle non-trivial arithmetic/logic relations across multiple input elements, it has limitations in dealing with other relations commonly present in complex real world programs. Particularly, arithmetic relations are often induced by data dependencies, and resolving constraints on such relations requires reasoning about input value changes, which symbolic execution is very good at. However, in practice there are other more subtle correlations pertain to field offset, and file/field length, which require reasoning about fields relocation, length variation, and fields removal and duplication.

Consider check ⓔ, whose left-hand side (*lhs*) is affected by the values (*cd_size* and *cd_off*) extracted from the input file, and the right-hand side (*rhs*) affected by the offset (*eocd_off*) which is not derived from input bytes through arithmetic operations, but rather from the branch outcomes of loop predicate at line 06. Such a check can be satisfied by either decreasing the lhs value or increasing the rhs value. In general, it is difficult for symbolic execution to directly reason-about/mutate the value of *eocd_off*. Hence, it is more likely that the lhs variables are set to a small value to satisfy check ⓔ. This would result in the failure of check ⓕ. In fact, we used KLEE to symbolically execute LibZip for 24 hours and found that check ⓕ is always unsatisfiable, preventing KLEE from further path exploration.

There has been proposal to use symbolic variables to model loop counts and then leverage program analysis to identify linear relations between loop count variables and input features such as length [23]. While modeling linear relations between loop count and input length is particularly effective in reasoning about length changes to overflow buffers, such relations are only one kind of the non-arithmetic/non-logical relations. For example, it cannot handle the relation involving `eocd_off` in check ⓔ, which denotes the offset of a particular input field instead of simple length.

Symbolic execution also has difficulty in modeling constraints derived from complex computation such as those involved in checksum (e.g., check ⓗ). In addition, many symbolic execution tools require generating models for library calls, entailing substantial manual efforts in practice.

Our Technique. We develop a novel fuzzing technique SLF that can effectively generate valid seed inputs from scratch. It is mutation fuzzing based so that it inherits the benefits of fuzzing. In addition, it overcomes the limitation of existing fuzzing techniques regarding seed input reliance and is capable of effectively composing valid seed inputs. SLF is inspired by two important observations. First, *input validation checks can be classified into a number of categories, each corresponding to specific input relations and entailing unique mutation strategies.* In our classification, gradient guided fuzzing [14] and symbolic execution are good at certain categories but not sufficiently general to deal with all the categories. Second, *the categories can be effectively detected during fuzzing by mutating input bytes in pre-defined fashions and observing the changes at predicates.* Such runtime detection can piggyback on a vanilla fuzzing infrastructure such as AFL. Our technique does not require any heavy-weight analysis such as taint-analysis. For example, by monitoring check ⓔ, we can

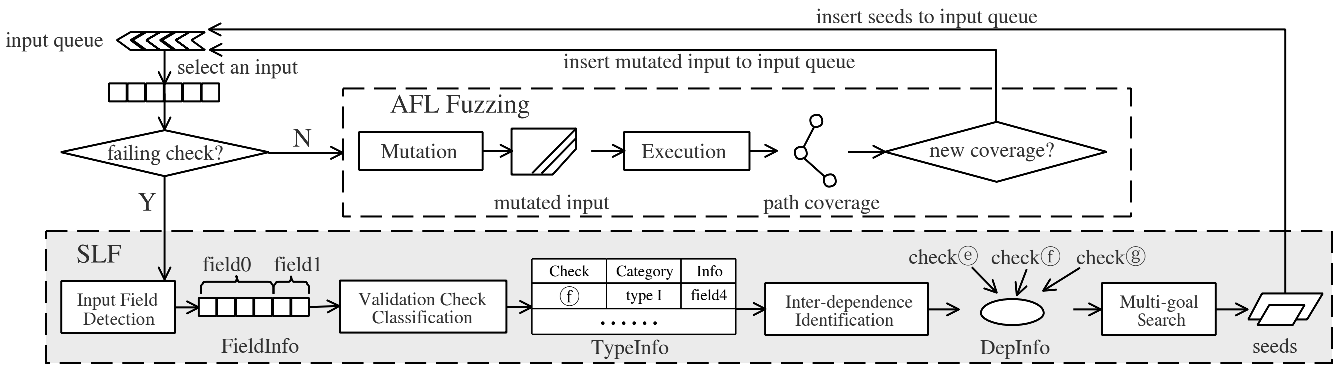


Fig. 3: Overview of SLF.

observe that the value of the lhs variable changes if we mutate certain input bytes, indicating likely arithmetic/logic relations, and the value of rhs changes if we add extra bytes to the head of input file, indicating index-of relations.

Based on the observations, we develop SLF. It starts from a random input of a few bytes. While the input most likely fails some validation check very shortly after the execution starts, our technique mutates the input and observes how the predicate corresponding to the check changes. This is to determine the category of the check and the correlated input byte(s). Based on the type of check, it applies the corresponding mutation/search strategy. It is very common that the input bytes that are identified as influencing the check may involve in some preceding checks that the execution has succeeded. As such, mutating them may lead to undesirable failures at those preceding checks. SLF features the capabilities of detecting inter-dependent checks that may be affected by mutating the same input bytes, and a multi-goal gradient guided search algorithm that aims to satisfy all these checks.

For our motivation example, our tool starts with a 4-byte random input, within 15 minutes, it generates a valid seed input as shown in Figure 1 that passes all the input validation checks. In 24 hours, it generates 7 valid seed inputs, which are further leveraged by regular fuzzing to cover 210 paths. In contrast, neither KLEE nor AFL generates any valid input. None of the test cases generated by KLEE can pass through check ⓕ. AFL does not pass through check ⓕ either. Most of the paths explored by AFL are exception-handling code. The comparison with other tools such as S2E, AFLFast, and Driller shows similar improvement (see Section IV-B).

III. Design

A. Overview

Figure 3 presents the overview of SLF, which is built on top of AFL. It consists of the following components. The first one is an input field detection component that groups input bytes to fields. This allows mutation to be performed at the unit of fields instead of bytes. The second one is a validation check classification component, which detects the types of validation checks in the current execution. The third one is an inter-dependence identification component, which identifies all the preceding checks that have inter-dependencies with the check that fails the current execution. Based on the results from the second and third components, the forth component, a multi-goal gradient-based search algorithm performs corresponding mutations to satisfy all the inter-dependent checks.

The overall procedure of SLF is presented in Algorithm 1. The input queue is initialized with a randomly-generated 4-bytes input (lines 1-2). In the main loop, SLF executes the target program with the current input (line 4), analyzing the encountered checks and error information to identify the check that fails the execution (line 5). If no validation failure is found, which indicates the current input is well-formed, SLF switches to regular AFL fuzzing (line 13). Otherwise, SLF tries to get through the failed check as follows.

First, it groups input bytes into fields (line 7). This is achieved by observing how the mutation of individual bytes affects the encountered checks. Consecutive bytes whose mutation affects the same set of checks are grouped to an input field, as detailed in Section III-B. Given the field information, SLF then classifies the encountered checks (line 8), by mutating input fields in predefined manners (e.g. flipping input field value and duplicating input field), as detailed in Section III-D. After that, SLF identifies all the checks that are correlated to the failed check, that is, they are influenced by the same input field(s) (Section III-E). A multi-goal search algorithm is then applied to satisfy these checks (Section III-F). Finally, if SFL succeeds in generating valid seeds, it adds them to the input queue for further processing.

SLF does not require source code. It runs the program executable on a modified QEMU to monitor the lhs and rhs values of `cmp` and `test` instructions that correspond to the conditional jump instructions.

B. Input Field Detection

The goal of input field detection is to group consecutive bytes that affect the same set of checks to a field. Algorithm 2 describes the process. We first execute the target program over the given input and collect the information of the encountered checks including their lhs and rhs values, denoted as $CheckInfo$ (line 2). Then we flip each input byte (line 5), that is, flipping individual bits in the byte, and execute the target program over the flipped input to collect new check information, denoted as $CheckInfo'$ (line 6). By

Algorithm 1: SLF's fuzzing loop.

```
if InputQueue = ∅ then
    InputQueue ← {RandomBytes(4)}
for input ∈ InputQueue do
    CheckInfo ← RunProgram(input)
    fcheck ← GetFailingCheck(CheckInfo)
    if fcheck ≠ null then
        FieldInfo ← GroupBytes(input)
        TypeInfo ← ClassifyChecks(input, FieldInfo)
        DepInfo ← GetDependents(fcheck, TypeInfo)
        seeds ←
          MultiGoalSearch(fcheck, TypeInfo, DepInfo)
        if seeds ≠ null then
            AddToQueue(InputQueue, seeds)
    AFLFuzzing(program, input)
```

Algorithm 2: $GroupBytes(input)$: input field grouping.

```
1  FieldInfo ← []
2  CheckInfo ← RunProgram(input)
3  CheckDiffPrv ← ∅
4  for byte ← 0 to len(input) do
5      input' ← FlipByte(input, byte)
6      CheckInfo' ← RunProgram(input')
7      CheckDiffCur ← Diff(CheckInfo, CheckInfo')
8      if CheckDiffCur = CheckDiffPrv then
9          FieldInfo ← UpdateLastGroup(FieldInfo, byte)
10     else
11         FieldInfo ← NewGroup(FieldInfo)
12         CheckDiffPrv ← CheckDiffCur
```

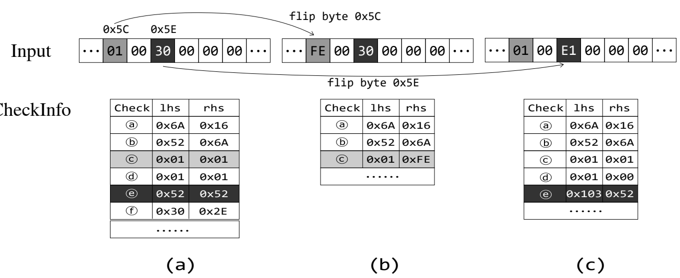


Fig. 4: Example of input bytes grouping.

comparing the difference of $CheckInfo$ and $CheckInfo'$, we can identify those checks that appear in both executions and have different lhs or rhs values (line 7). The two pieces of information are aligned based on the program counters. If the difference caused by flipping the current byte is the same as that of the previous byte, they are grouped together (line 9). Otherwise, a new field is created (line 11). Note that in the case where bytes [i-1, i] affect one check and bytes [i, i+1] affect another, SLF treats each byte as a separate group, since they have different affected check sets.

Figure 4 illustrates how the bytes at index 0x5C-61 are grouped. Figure 4a shows the $CheckInfo$ of the execution on the input, which records the lhs and rhs values of the encountered checks. Particularly, the rhs value of check c is 0x01 and the lhs value of check e is 0x52. After flipping the byte at 0x5C, the $CheckInfo'$ of the mutated execution is shown in Figure 4b. The difference between $CheckInfo$ and $CheckInfo'$ is that the rhs value of check c changes. Similarly, after flipping the byte at 0x5D, we get the same difference. In contrast, flipping the byte at 0x5E results in the difference at the lhs value of check e. Hence, we group the bytes at 0x5C-5D as a field and byte 0x5E starts a new field. An important feature of the algorithm is that it does not require the input to be a valid one to begin with.

C. Input Validation Check Classification

To facilitate discussion, we introduce a language to model input validation checks, which allows us to classify such checks based on the underlying relations with input elements. Figure 5 shows the syntax of the language. For simplicity, the language operates at individual input bytes (while our

$$
\begin{array}{llcl}
\langle Expression\rangle & e & ::= & input[lb, ub] \mid c \mid x \\
 & & \mid & e_1 \ \texttt{binop}\ e_2 \mid e_1 \ \texttt{relop}\ e_2 \\
 & & \mid & \texttt{F}(e_0) \mid \texttt{ITE}(e_p, e_t, e_f) \\
 & & \mid & \texttt{count}(input[lb, ub], e_p) \\
 & & \mid & \texttt{indexOf}(input[lb, ub], e_p) \\
\langle Check\rangle & C & ::= & \texttt{assert (e)} \\
\langle Input\rangle & input & ::= & \texttt{byte}^\star \\
\langle Constant\rangle & c & ::= & \{\texttt{'true'}, \texttt{'false'}, 0, 1, ...\} \\
\langle BinOperator\rangle & binop & ::= & + \mid - \mid * \mid / \mid ... \\
\langle RelOperator\rangle & relop & ::= & == \mid != \mid > \mid ...
\end{array}
$$

Fig. 5: Language

implementation deals with fields). We use $input[lb, ub]$ with lb the lower bound and ub the upper bound to denote an input segment. Note that $input[0, len(input)]$ denotes the entire input. Observe that an expression could be an input segment, a constant, a variable x whose use will be explained later in this paragraph, binary arithmetic operation and relational operation of expressions, a function *F()* on a parameter e_0 to denote an uninterpreted function whose internals are not visible/understandable, an *If-Then-Else* (*ITE*) expression whose value may be e_t or e_f, depending on if e_p is true, a *count()* primitive that counts the number of cases in an input segment that satisfy a condition e_p, and an *indexOf()* primitive that identifies the offset of the first case that satisfies e_p. Variable x is used in e_p to denote some byte in the given input segment. For example, $count(input[0, len(input)], x > 10)$ counts all the input bytes whose value is larger than 10. A validation check asserts an expression to be true. Note that we do not model program variables in our language and expressions are essentially formulas on input segments and constants.

The language is expressive to describe most of the validation checks we have observed in our subject programs. For example, check h is an instance of *F()*, where F stands for the checksum function that a symbolic execution tool has difficulty modeling. Other examples of *F()* include libraries without source code. Expression $count(input[cmt\_off, len(input)], true)$ models the predicate variable *tail_len* in check g, which counts the bytes from the comment offset (*cmt_off*) to the end of file. Expression $indexOf(input[0, len(input)],\ x == MAGIC\_EOCD)$ models the *eocd_off* predicate variable in check b, which means to find the offset of the `eocd` magic number.

The formalization allows us to classify validation checks. In

Algorithm 3: $ClassifyChecks(input, FieldInfo)$: detecting check types.

$TypeInfo \leftarrow []$
$CheckInfo \leftarrow RunProgram(input)$
for $field \in FieldInfo$ **do**
  for $type \in TypeList$ **do**
    $policy \leftarrow GetDetectPolicy(type)$
    $input' \leftarrow EnforcePolicy(input, policy)$
    $CheckInfo' \leftarrow RunProgram(input')$
    $CheckDiff \leftarrow Diff(CheckInfo, CheckInfo')$
    for $check \in CheckDiff$ **do**
      if $MatchPolicy(check, policy)$ **then**
        $info \leftarrow ClassifyType(check, type)$
        $TypeInfo[check] \leftarrow info$

the following, we discuss four most popular categories, each corresponding to some form of expression. The classification is designed in a way to facilitate later fuzzing.

Type I: Arithmetic Check. If the variables involved in a check are derived from input bytes through only the arithmetic operations (e.g., the `binop` in our language), the check is a type I arithmetic check. Symbolic execution is particularly good at satisfying type I checks. Gradient based fuzzing [14] is also good when there are no other inter-dependent checks. Checks ⓒ and ⓓ in Figure 2 are type I checks.

Type II: Index/Offset Check. Index checks have the form of $assert(indexOf(input[lb, ub], e_p)\ relop\ e_0)$ that compares the offset of the first case satisfying e_p in the given input segment with another expression e_0. They are also quite common but difficult to satisfy by existing techniques. Check ⓔ in Figure 2 is a type II check.

Type III: Count Check. Count checks have the form of $assert(count(input[lb, ub], e_p)\ relop\ e_0)$ that compares the number of cases satisfying e_p in the given input segment with an expression e_0. All the length checks, checks that count the number of data structures/objects, fall into this category. They are commonly present but difficult to satisfy by existing techniques. Check ⓖ in Figure 2 is a type III check.

Type IV: ITE Check. ITE checks have the form of $assert(ITE(e_p, e_t, e_f)\ relop\ e_0)$, in which e_p is an expression that are based on input bytes through only arithmetic operations. Check ⓐ in Figure 2 is a type IV check. Observe that the result of the check has control dependence on the input bytes instead of data dependencies. Our technique resorts to the default random mutation scheme of AFL to reason about such checks. Note that a more complex solution may entail heavy-weight program analysis that affects the applicability of our technique. We leave it to our future work to develop a more sophisticated solution.

While these categories cover the checks in the programs we consider, we do not claim their comprehensiveness. Moreover, they do not represent a strict partitioning and hence a check may fall into more than one categories. For example, the lhs of check ⓔ in Figure 2 suggests type I and the rhs type II.

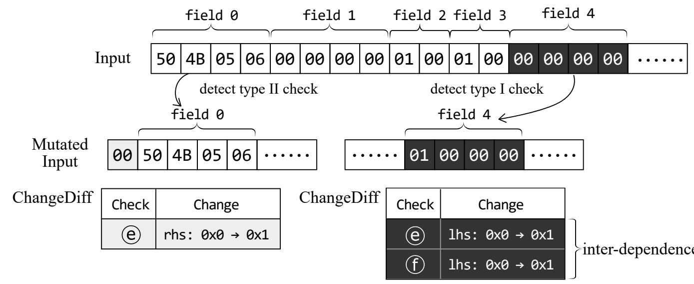


Fig. 6: Type checking and inter-dependence detection.

D. Detecting Check Types

Due to the different features of each type of check, we define different policies to detect each of them. Algorithm 3 describes the process of type detection. With the field information got from the previous step, we iterate over individual input fields and try the predefined detection policy for each type. We classify a check to a type by examining whether the change of its predicate variable values match the feature of the type. The order of type detection attempts does not matter. Currently, we support the detection policies for types I, II, III.

Detecting Type I. For each field under consideration, we add a random value to the field. We then examine whether the mutation results in the value of a certain predicate variable changed. If so, we mark the corresponding check as an arithmetic check. Intuitively, it leverages the observation that most arithmetic operations in programs denote one-to-one mapping from input to output. In other words, any change of the input leads to change of output. There are corner cases such as mod operation that does not denote a one-to-one mapping. SLF currently does not handle them. We also record the field that affects the check. The subsequent multi-goal search algorithm relies on such information.

Detecting Type II. To detect type II checks, we insert an extra byte before the current field under consideration. We examine whether the extra byte results in the value of a certain predicate variable off by one. If so, we mark the corresponding check as index/offset check. We also record the field before which we insert the padding byte.

Detecting Type III. To detect Type III checks, the idea is to duplicate the object/data-structure that is being counted and observe if any variable at the check changes. The challenge lies in that we do not know the boundaries of objects/structures. Note that an object/structure may consist of multiple fields. We resort to a search algorithm. Assume the input has n fields in total. For each field i under consideration in Algorithm 3, we start a loop j from i to n. In the loop, we duplicate the input segment including fields from i to j and insert the duplicated segment after field j. Essentially, we are considering fields i-j form an object. We then examine whether the insertion results in variable value changes at the check, which indicates a type III check. We also record i and j.

Example. Figure 6 presents an example to illustrate check type detection. Assume we are given an invalid input as shown in the top of the figure. When we iterate on field 0 and try the detection policy for type II, an extra byte is inserted before field 0. Such a mutation results in the rhs of check e increasing by one, which matches the feature of type II. For field 4, when we increase its value by 1, the lhs of check e changes. It matches the feature of type I check. We mark check e as type I due to lhs and record that it is affected by field 4, and also type II due to rhs.

E. Identifying Inter-dependent Checks

Given the type information of each check, we correlate those checks whose predicates are affected by the same field. For example, the input in Figure 6 will fail check f. The information provided by the check type detection phase shows that checks e and f are affected by the same field (i.e., field 4). We hence correlate them for further processing.

F. Gradient-based Multi-goal Search Algorithm

This step aims to identify mutation that is likely to pass the failing check and does not fail any preceding checks. It starts from the current failing check and the corresponding input, and mutates the input based on the type and inter-dependence information of the failing check. Algorithm 4 presents part of the procedure. For simplicity, we assume a singular inter-dependent check rc and a singular field fd that affects the failing check fc. Our implementation handles multiple inter-dependent checks and multiple fields affecting fc.

In the first case (lines 1-14), the lhs of fc indicates that it is a type I check while the rhs is a constant value. At lines 4-5, the program is re-executed with the mutated field fd'. The mutation is bounded random mutation. The new runtime information $CheckInfo'$ contains the updated information of the failing check, denoted as fc'. At lines 6 and 7, the gradients of the failing check and the relevant check are computed. Observe that they denote how the mutation on fd affects the difference of the lhs and rhs at the check. Line 8 tries to select a field value that can negate the failing check but still respect the relevant check, by choosing a value that is greater than $fd - (fc.lhs - fc.rhs)/gradient$, which would change the branch outcome at fc if it denotes a linear function, and smaller than $fd - (rc.lhs - rc.rhs)/relgrdt$ so that rc likely retains its branch outcome. If the range at line 8 is empty and fd'' cannot be selected, indicating potential infeasibility of satisfying both checks. Lines 9-13 choose to first satisfy the check that has less flexibility (i.e., influenced by smaller number of other input fields and hence smaller room to manipulate) while getting closer to satisfy the other check. Specifically, line 11 sets the mutated value to one that is slightly smaller than a value that would likely negate the branch outcome of rc; line 13 negates fc with the smallest possible value so that the violation of rc may likely be minimized. The resulted seed input is added to the set.

Lines 15-21 denote another case in which the lhs indicates type I whereas the rhs indicates type III. Since both sides can be manipulated by input mutations, the strategy of SLF is to fix one side and mutate the other side. It generates two seed inputs: one by fixing rhs and mutating lhs and vice versa. The search space of mutating both sides is explored by further mutating the two new seeds (in later rounds). This strategy is particularly effective for passing checksum validity checks. Since lhs is type I, mutating lhs is similar to that in lines 4-14 and elided. Lines 17-21 are to fix lhs and mutate rhs. In particular, since rhs indicates type III, line 18 extracts the starting and ending field indexes of the object/structure that is being counted. It then mutates the input by duplicating the object/structure once and adding it right after the ending index ub, and re-executes the program (line 19). A gradient value is computed to measure the influence at the failing check. Note that the denominator is 1 as we only add one object. Line 21 tries to insert $N = gradient$ objects to negate the failing check. Note that when a failing type II/III check is inter-dependent to other type I checks, SLF always tries to negate the type II/III check first and lets the following rounds to fix the possible violations of the inter-dependent checks.

Algorithm 4: $MultiGoalSearch(fc, TypeInfo, DepInfo)$: gradient-based multi-goal Search. Variables $input$, fc, rc, fd, $seeds$ denote the current input, the failing check, the inter-dependent check, the field that impacts fc's variables, and inputs generated, respectively.

1 **if** $fc.lhs.cat == I \;\&\& fc.rhs.cat == CONST$ **then**
2 $\quad fd \leftarrow GetDepField(TypeInfo, fc.lhs)$
3 $\quad rc \leftarrow GetDepCheck(DepInfo, fd)$
4 $\quad fd' \leftarrow MutateField(fd)$
5 $\quad CheckInfo' \leftarrow RunProgram(ReplaceField(input, fd, fd'))$
6 $\quad gradient \leftarrow \frac{(fc'.lhs - fc'.rhs) - (fc.lhs - fc.rhs)}{fd' - fd}$
7 $\quad relgrdt \leftarrow \frac{(rc'.lhs - rc'.rhs) - (rc.lhs - rc.rhs)}{fd' - fd}$
8 $\quad fd'' \leftarrow$ a value in $[fd - (fc.lhs - fc.rhs)/gradient,\ fd - (rc.lhs - rc.rhs)/relgrdt)$
9 $\quad$ **if** $fd'' == null$ **then**
10 $\quad\quad$ **if** fc *has more flexibility than* rc **then**
11 $\quad\quad\quad fd'' \leftarrow fd - (rc.lhs - rc.rhs)/gradient + \delta$
12 $\quad\quad$ **else**
13 $\quad\quad\quad fd'' \leftarrow fd - (fc.lhs - fc.rhs)/gradient$
14 $\quad seeds \leftarrow AddSeed(seeds, ReplaceField(input, fd, fd''))$
15 **if** $fc.lhs.cat == I \;\&\& fc.rhs.cat == III$ **then**
16 $\quad$ /*fix rhs and mutate lhs like lines 4-14 */
17 $\quad$ /*fix lhs and mutate rhs*/
18 $\quad \langle lb, ub\rangle \leftarrow GetDepField(TypeInfo, fc.rhs)$
19 $\quad CheckInfo' \leftarrow RunProgram(InsertFields(input, up + 1, Fields[lb, ub]))$
20 $\quad gradient \leftarrow (fc'.lhs - fc'.rhs) - (fc.lhs - fc.rhs)$
21 $\quad seeds \leftarrow AddSeed(seeds, InsertNFields(input, ub + 1, Fiels[lb, ub], gradient)$
22 ...

There are other cases. Due to the space limitations, we omit them from discussion. They are similarly handled as the aforementioned two cases.

Example. Assume check f in Figure 1 fails. It belongs to the first case in Algorithm 4. Further assume that cd_size, cd_offset, $SIZE\_CDH$, and $eocd\_off$ are 0x2f, 0x22, 0x33, and 0x51, respectively. SLF identifies checks e and f inter-dependent. SLF determines the gradients for both e and f are 1 (by mutating cd_size and comparing lhs−rhs before and after mutation). However, it fails to choose a value that satisfies ⓕ without failing ⓔ, as $cd\_size + cd\_off - eocd\_off == 0$ while $cd\_size - SIZE\_CDH == -0x4$. In this case, SLF chooses to satisfy the more restricted check ⓕ by adding 4 to cd_size. In a later round, the input is further mutated by changing cd_off to pass all checks.

G. Limitations

SLF cannot handle checks that have complex control dependencies with input fields and hence relies on regular fuzzing to deal with these fields. In addition, SLF identifies input fields at the byte level and hence cannot reason about checks that involve bit-level fields.

IV. Evaluation

A. Experiment Setup

Our evaluation is performed on two datasets. One contains 10 real-world programs that are commonly used in other fuzzing projects [17], [18]. The other is 10 benchmark programs selected from the Google fuzzer test-suite [19] that do not have valid seed inputs provided. These programs cover a wide range of categories, including image, audio, compression, font, etc. The first three columns in Table I and Table II present detailed information of the programs.

We compare our tool (SLF) with other popular state-of-the-art testing tools. For symbolic execution tools, we compare with KLEE [9] and S2E [13]. Due to the input size needed in our programs, we provide a 256-bytes symbolic file as input for both KLEE and S2E. We use the coverage-optimized search strategy for KLEE and the class-uniform analysis search strategy for S2E, which are the recommended settings. For mutation based fuzzers, we compare with the original AFL [16] and its improved version AFLFast [20] that performs optimized fuzzing energy allocation. For hybrid fuzzers, we compare with Driller [15], which combines fuzzing with symbolic execution (for exploring paths difficult to fuzz).

All of our experiments are performed on a machine with 12 cores (Intel® Core™ i7-8700 CPU @ 3.20GHz) and 16 GB memory running the Ubuntu 16.04 operating system.

B. Results for the Programs Commonly Used in Fuzzing

To evaluate the effectiveness of SLF, we perform a set of experiments by running SLF and other tools on the 10 real-world programs commonly used in fuzzing with a 4-byte randomly generated invalid input, and observing their path coverage changes over time. We run each experiment (one testing tool on one program) for 24 hours. Results are presented in Table I and Figure 7, in which X-axis represents time, Y-axis represents the number of covered paths for each experiment, and the points on the curves denote the moments that a valid seed input is generated.

From these results, we can make the following observations. Firstly, traditional mutation-based fuzzers (i.e., AFL and AFLFast) do not work well without valid inputs in most cases. At the beginning, they can cover some paths by satisfying environment related path conditions but very few or no input-related conditions. After that, the executions got stuck at finding the first valid input and cannot proceed. In many cases, it cannot make any progress within 24 hours, leading to low path discovery. For example, in the evaluation of libtiff (Figure 7c), AFL and AFLFast got stuck within 2 hours, yielding only 1/6 path coverage compared with SLF. As shown in Table I, on average SFL covers 3.0 and 2.5 times more paths than AFL and AFLFast, and generates much more valid seeds.

Secondly, the symbolic execution tools work well on small programs. For example, S2E achieves the best result than all the fuzzing-based tools on otfcc (Figure 7j). However, they do not handle complex real-world programs well. We have spent substantial engineering efforts to make KLEE and S2E to work on these target programs, even under the guidance from some of their original developers. We were able to make KLEE run on 7 target programs and S2E on 9 target programs. Those programs that cannot run for KLEE/S2E are marked as `N/A` in Table I. Also note that S2E is very memory-consuming. If the memory is about to reach the limit, it starts to randomly kill some of the existing states until there is enough memory for continuation. Due to the random state killing, it often terminates early as there are no more states to explore. It runs for 5 hours on average in our experiments. We mark the termination point of S2E with an "x" symbol in Figure 7. For programs that can be run by KLEE, SLF generates 1.6 times more seeds than KLEE in total; for those programs that can be run by S2E, SLF generates 6.4 times more seeds. SFL outperforms symbolic execution except for the three smallest programs (i.e., giflib, otfcc, ttf2woff).

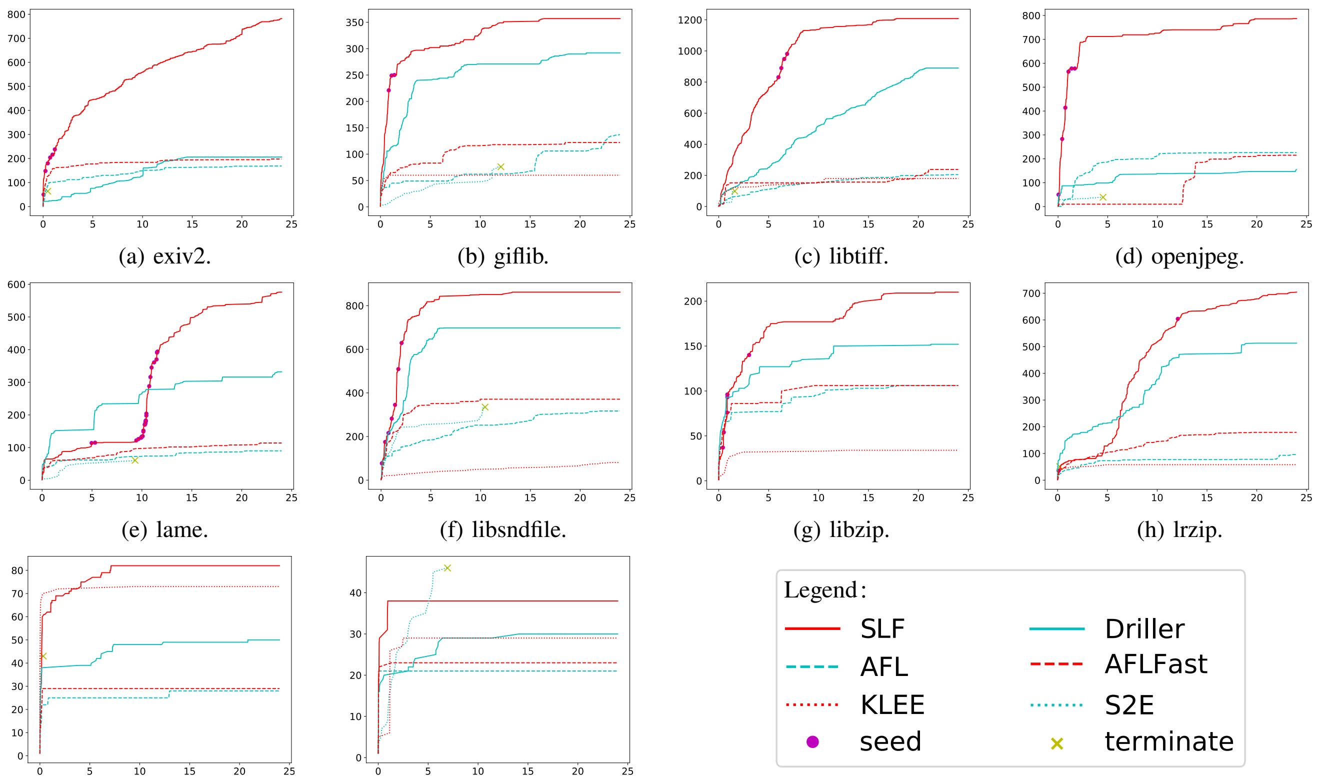
Thirdly, the hybrid fuzzer Driller alternates between fuzzing (in most of the time) and symbolic execution (when fuzzing is stuck). It scales to large programs and achieves reasonable performance. However, it still does not work as well as our tool. From our manual inspection, the reason seems to be that the optimal timing of switching from fuzzing to symbolic execution is difficult to identify for complex programs. Switching at a wrong time does not really exploit the benefits of the combination. Furthermore, there are still cases that the symbolic execution engine cannot handle well.

Lastly, our tool SLF achieves the best performance (i.e., the largest number of explored paths) in all cases except otfcc. As we can see from the graph, newly generated seed inputs usually lead to exploration of a large number of new paths. These results show that SLF is very effective in generating new valid seeds, which lead to better path coverage than other tools in most cases.

Case Study. We use Exiv2 as a case study. Within 24 hours, SLF generates 6 unique seeds that lead to the coverage of 782 different program paths, while other tools do not generate any valid seed. SLF has more than three times path coverage than other tools. More interestingly, we found that one of the generated seeds has undocumented format. Figure 8 shows the snippets of a documented valid seed and the undocumented valid seed. In the documented format, the bytes at offset

TABLE I: Evaluation result on real-world programs.

Category	Program	SLOC	SLF		AFL		AFLFast		Driller		KLEE		S2E	
			Paths	Seeds	Paths	Seeds	Paths	Seeds	Paths	Seeds	Paths	Seeds	Paths	Seeds
Image	exiv2	191,993	782	6	169	0	198	0	206	0	N/A	N/A	64	0
	giflib	8,209	357	3	137	1	122	1	292	1	60	4	76	2
	libtiff	82,484	1,208	4	205	0	238	0	890	0	180	0	99	0
	openjpeg	164,284	787	6	226	0	215	0	155	0	N/A	N/A	39	0
Audio	lame	60,240	576	31	90	0	114	0	332	9	N/A	N/A	61	0
	libsndfile	70,064	862	7	317	4	356	4	698	3	81	0	335	5
Compression	libzip	17,985	210	7	106	0	106	0	152	0	34	0	N/A	N/A
	lrzip	19,098	704	2	96	0	179	0	513	0	58	0	51	0
Font	otfcc	10,344	38	0	21	0	23	0	30	0	29	4	46	1
	ttf2woff	5,565	82	0	28	0	29	0	50	0	73	1	43	0



(a) exiv2. (b) giflib. (c) libtiff. (d) openjpeg.

(e) lame. (f) libsndfile. (g) libzip. (h) lrzip.

(i) ttf2woff. (j) otfcc.

Fig. 7: Path coverage. X-axis: time over 24 hours, Y-axis: the number of unique paths.

0x04-07 is an offset field that points to the location where the subsequent data (e.g., count and tag) are stored. Usually, the offset points to the following bytes. However, in the undocumented format, the offset points to itself. As a result, the subsequent data overlaps with the offset field. Hence the count is 0x04 and the tag is 0x00. Such seed is uniquely supported by Exiv2 and not supported by other TIFF image processing tools (e.g., LibTiff). This seed input yields 86 new path coverage in the fuzzing process.

We illustrate how SLF generates such undocumented seed. Figure 9 shows two critical checks in Exiv2. During fuzzing process, an 8-byte invalid input that fails the two checks is given to SLF. Check ① is of type I (lhs) and type II (rhs). Our multi-goal search algorithm will try to fix rhs (i.e., file_size) and mutate lhs (i.e., input[0x04, 0x07]) to satisfy the check assertion (line 16 of Algorithm 4), which results in assigning input[0x04, 0x07] with value 0x04. Check ② is of type I (lhs) and type II (rhs) and inter-dependent with check ① on input[0x04, 0x07]. To satisfy both checks, our search algorithm will try to extend the file_size. After satisfying other subsequent checks, the undocumented seed is generated.

C. Google Fuzzer Test Suite

Another set of experiments is on the Google fuzzer test suite. As a standard fuzzing benchmark, it tags some locations to see if a testing tool can reach those locations. Table II shows the programs, the target location in the program and time used to reach this location (measured by hours). We only use the programs that do not come with a valid seed. In total, we evaluate on 18 locations. In the table, `T/O` means time out after 24 hours and `N/A` means the tool does not work on this program. S2E does not work on this benchmark. We have confirmed that with the authors of S2E. The probable reason is that the target programs or the underlying third-party libraries contain instructions operating on SSE, MMX and FP registers, which are currently not supported by S2E.

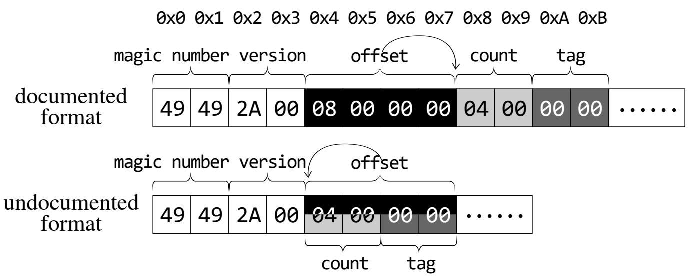


Fig. 8: Example of undocumented TIFF format.

```
offset = input[0x04, 0x07];
if (offset + 4 > file_size) error();                ①
count = input[offset, offset+1];
if (12 * count + offset > file_size) error();       ②
```

Fig. 9: Two critical checks in Exiv2.

In summary, our tool ran on all the programs and can reach 9 locations with reasonable execution time. AFL, AFLFast and Driller also work on all programs, and each can reach 4 locations within 24 hours. KLEE does not work on harfbuzz as some of its instructions are not supported by KLEE. In total, KLEE is able to reach 6 locations. SLF is able to reach all the locations reached by any other tools. In terms of time used to reach the location, it outperforms AFL, AFLFast and Driller in all cases. In some cases, KLEE takes less time. Manual inspection discloses that these locations are on shallow paths. For complex cases (e.g., lcms) where KLEE cannot reach within 24 hours, SLF is still able to reach the location.

D. Threats to Validity

The initial input may impact the effectiveness of fuzzing. To be fair, we used the same 4-byte invalid input for AFL, AFLFast, Driller, SLF. For symbolic execution tools, the symbolic file size and path exploration algorithm impact results. We used the recommended settings for S2E and KLEE.

V. Related Work

Mutation based fuzzing. Mutation based fuzzing generates new inputs by mutating seed inputs. The random nature of mutations often leads to lengthy fuzzing time. A number of existing works were proposed to boost fuzzing by improving seed selection [24], optimizing fuzzing guidance using statistical metrics [20], [25]–[27], and leveraging vulnerability specific patterns [28]. These techniques mainly aim to improve the quality of generated/selected seeds for fuzzing. This demonstrates the importance of seed inputs. Compared to these techniques, SLF does not require a valid seed to begin with. It has the unique feature of satisfying input validity checks.

Generative fuzzing. Another popular fuzzing method is to generate valid seed inputs from input format specifications. Some approaches, e.g., SPIKE [29] and Peach [30], take human-written input specifications as templates. Other approaches, e.g., Skyfire [5] and Learn&Fuzz [6] learn specification from a large corpus. They are effective when a high quality training set is available.

TABLE II: Evaluation result on Google fuzzer test-suite.

Program	SLOC	Location	Reaching Time (hours)				
			SLF	AFL	AFLFast	Driller	KLEE
freetype2	168,231	ttgload.c:1710	T/O	T/O	T/O	T/O	T/O
guetzli	8,068	output_image.cc:398	5.50	T/O	T/O	T/O	T/O
harfbuzz	13,099	hb-buffer.cc:419	T/O	T/O	T/O	T/O	N/A
lcms	55,918	cmsintrp.c:642	8.27	T/O	T/O	T/O	T/O
libarchive	190,125	archive...warc.c:537	T/O	T/O	T/O	T/O	T/O
libjpeg	57,437	jdmarker.c:659	0.16	0.31	5.73	0.20	0.08
libpng	20,820	png.c:1035	4.55	T/O	T/O	T/O	0.02
		pngread.c:757	0.49	T/O	T/O	T/O	1.83
		pngrutil.c:1393	T/O	T/O	T/O	T/O	T/O
		pngread.c:738	0.49	T/O	T/O	T/O	0.02
		pngrutil.c:3182	T/O	T/O	T/O	T/O	T/O
		pngrutil.c:139	0.01	0.03	0.01	0.01	0.01
proj4	37,871	PJ_urm5.c:39	0.82	4.47	3.26	2.73	T/O
vorbis	61,990	codebook.c:479	T/O	T/O	T/O	T/O	T/O
		codebook.c:407	T/O	T/O	T/O	T/O	T/O
		res0.c:690	T/O	T/O	T/O	T/O	T/O
woff2	34,594	woff2_dec.cc:500	T/O	T/O	T/O	T/O	T/O
		woff_dec.cc:1274	0.45	1.37	1.26	1.07	0.02

Symbolic execution. Symbolic execution techniques are very popular and effective software testing [10]–[12], [31]–[42]. Like SLF, it does not require seed inputs to begin with. Fuzzing and symbolic execution both have their pros and cons. For example, fuzzing is usually more applicable and easily works on large and complex software. SLF belongs to fuzzing and hence has similar benefits. While traditional fuzzers are inferior to symbolic execution in precise resolution of path conditions, SLF mitigates such limitations by having a gradient guided multi-goal search algorithm. Additionally, SLF features the capabilities of directly resolving the constraints in various kinds of validity checks, such as offset and count checks.

Search-based testing. Search-based testing techniques view testing problems, including automatic generation of test cases, test case minimization and prioritization, and regression testing as search problems, and try to use meta-heuristic search techniques (e.g., genetic algorithms, simulated annealing and tabu search) to solve them [43]–[56]. In particular, EVOSUITE [57] optimizes a test suite towards satisfying a coverage criterion. While fuzzing can be viewed as a search based testing technique, SLF uniquely focuses on validity checks and works by classifying such checks and performing class-specific mutations. On the other hand, SFL may be complementary to existing search-based techniques by providing field information and validity check type information.

Hybrid fuzzing. Hybrid fuzzing performs fuzzing for most of the time and performs symbolic execution when needed. Driller [15] and Angora [14] are both such fuzzers. They can handle large real-world programs (using fuzzing) as well as leveraging the power of symbolic execution to solve difficult constraints. However, when to start the symbolic execution remains a hard problem for such tools (as shown in Section IV).

VI. Conclusion

We develop a novel fuzzing technique that can generate valid seed inputs from scratch. It works by classifying input validity checks into a number of types and conduct type specific mutations. It features a multi-goal search algorithm that can satisfy multiple inter-dependent validity checks all together. Our evaluation shows that our technique is highly effective and efficient, outperforming existing tools.

Acknowledgment

The authors would like to thank the anonymous reviewers for their constructive comments. Also, the authors would like to express their thanks to Yang Xiao and Hui Peng for their help in experiment settings and Xinjie Wang for her help in illustration. The authors were supported in part by DARPA FA8650-15-C-7562, NSF 1748764 and 1409668, ONR N000141410468 and N000141712947, Sandia National Lab under award 1701331, and NSFC U1836209.

References

[1] "Oss-fuzz: Five months later, and rewarding projects," https://security.googleblog.com/2017/05/oss-fuzz-five-months-later-and.html.

[2] "How heartbleed could've been found," https://blog.hboeck.de/archives/868-How-Heartbleed-couldve-been-found.html.

[3] "Cisco webex .atp and .wrf overflow vulnerabilities," https://www.coresecurity.com/content/webex-atp-and-wrf-overflow-vulnerabilities.

[4] "Analysis and exploitation of pegasus kernel vulnerabilities," https://jndok.github.io/2016/10/04/pegasus-writeup/.

[5] J. Wang, B. Chen, L. Wei, and Y. Liu, "Skyfire: Data-driven seed generation for fuzzing," in *2017 IEEE Symposium on Security and Privacy, SP 2017*, 2017.

[6] P. Godefroid, H. Peleg, and R. Singh, "Learn&fuzz: machine learning for input fuzzing," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017*.

[7] J. Lee, T. Avgerinos, and D. Brumley, "TIE: principled reverse engineering of types in binary programs," in *Proceedings of the Network and Distributed System Security Symposium, NDSS 2011*, 2011.

[8] Z. Lin, X. Zhang, and D. Xu, "Automatic reverse engineering of data structures from binary execution," in *Proceedings of the Network and Distributed System Security Symposium, NDSS 2010*, 2010.

[9] C. Cadar, D. Dunbar, and D. R. Engler, "KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs," in *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*, 2008, pp. 209–224. [Online]. Available: http://www.usenix.org/events/osdi08/tech/full_papers/cadar/cadar.pdf

[10] X. Ge, K. Taneja, T. Xie, and N. Tillmann, "Dyta: dynamic symbolic execution guided with static verification results," in *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011*, R. N. Taylor, H. C. Gall, and N. Medvidovic, Eds. ACM, 2011, pp. 992–994. [Online]. Available: http://doi.acm.org/10.1145/1985793.1985971

[11] J. Chen, W. Hu, L. Zhang, D. Hao, S. Khurshid, and L. Zhang, "Learning to accelerate symbolic execution via code transformation," in *32nd European Conference on Object-Oriented Programming, ECOOP 2018, July 16-21, 2018, Amsterdam, The Netherlands*, ser. LIPIcs, T. D. Millstein, Ed., vol. 109. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2018, pp. 6:1–6:27. [Online]. Available: https://doi.org/10.4230/LIPIcs.ECOOP.2018.6

[12] S. Person, G. Yang, N. Rungta, and S. Khurshid, "Directed incremental symbolic execution," in *Acm Sigplan Notices*, vol. 46, no. 6. ACM, 2011, pp. 504–515.

[13] V. Chipounov, V. Kuznetsov, and G. Candea, "S2E: a platform for in-vivo multi-path analysis of software systems," in *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2011, Newport Beach, CA, USA, March 5-11, 2011*, 2011, pp. 265–278. [Online]. Available: http://doi.acm.org/10.1145/1950365.1950396

[14] P. Chen and H. Chen, "Angora: Efficient fuzzing by principled search," in *Proceedings of the 39th IEEE Symposium on Security and Privacy (SP 2018)*, 2018.

[15] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Driller: Augmenting fuzzing through selective symbolic execution," in *Proceedings of the 23rd Annual Network and Distributed System Security Symposium, NDSS 2016*.

[16] "American fuzzy lop (afl)," http://lcamtuf.coredump.cx/afl.

[17] S. Gan, C. Zhang, X. Qin, X. Tu, K. Li, Z. Pei, and Z. Chen, "Collafl: Path sensitive fuzzing," in *Proceedings of the 39th IEEE Symposium on Security and Privacy (SP 2018)*, 2018.

[18] "Oss-fuzz project," https://github.com/google/oss-fuzz/tree/master/projects.

[19] "Google fuzzer test suite," https://github.com/google/fuzzer-test-suite.

[20] M. Böhme, V. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as markov chain," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security CCS 2016*.

[21] "Libzip," https://libzip.org/.

[22] "Zip file format," https://en.wikipedia.org/wiki/Zip_(file_format).

[23] P. Saxena, P. Poosankam, S. McCamant, and D. Song, "Loop-extended symbolic execution on binary programs," in *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis, ISSTA 2009, Chicago, IL, USA, July 19-23, 2009*, 2009, pp. 225–236.

[24] A. Rebert, S. K. Cha, T. Avgerinos, J. Foote, D. Warren, G. Grieco, and D. Brumley, "Optimizing seed selection for fuzzing," in *Proceedings of the 23rd USENIX Security Symposium 2014*, 2014.

[25] C. Lemieux and K. Sen, "Fairfuzz: Targeting rare branches to rapidly increase greybox fuzz testing coverage," *CoRR*, vol. abs/1709.07101, 2017.

[26] T. Petsios, J. Zhao, A. D. Keromytis, and S. Jana, "Slowfuzz: Automated domain-independent detection of algorithmic complexity vulnerabilities," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017*.

[27] M. Böhme, V. Pham, M. Nguyen, and A. Roychoudhury, "Directed greybox fuzzing," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017*, 2017.

[28] I. Haller, A. Slowinska, M. Neugschwandtner, and H. Bos, "Dowsing for overflows: a guided fuzzer to find buffer boundary violations," in *Proceedings of the 22th USENIX Security Symposium Security*, 2013.

[29] "Github - guilhermeferreira/spikepp," https://github.com/guilhermeferreira/spikepp, (Accessed on 08/24/2018).

[30] "Github - mozillasecurity/peach: Peach is a fuzzing framework which uses a dsl for building fuzzers and an observer based architecture to execute and monitor them." https://github.com/MozillaSecurity/peach, (Accessed on 08/24/2018).

[31] S. Khurshid, C. S. Păsăreanu, and W. Visser, "Generalized symbolic execution for model checking and testing," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2003, pp. 553–568.

[32] C. Cadar, P. Godefroid, S. Khurshid, C. S. Pasareanu, K. Sen, N. Tillmann, and W. Visser, "Symbolic execution for software testing in practice: preliminary assessment," in *Software Engineering (ICSE), 2011 33rd International Conference on*. IEEE, 2011, pp. 1066–1071.

[33] S. Dong, O. Olivo, L. Zhang, and S. Khurshid, "Studying the influence of standard compiler optimizations on symbolic execution," in *26th IEEE International Symposium on Software Reliability Engineering, ISSRE 2015, Gaithersbury, MD, USA, November 2-5, 2015*. IEEE Computer Society, 2015, pp. 205–215. [Online]. Available: https://doi.org/10.1109/ISSRE.2015.7381814

[34] R. Qiu, G. Yang, C. S. Pasareanu, and S. Khurshid, "Compositional symbolic execution with memoized replay," in *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*, A. Bertolino, G. Canfora, and S. G. Elbaum, Eds. IEEE Computer Society, 2015, pp. 632–642. [Online]. Available: https://doi.org/10.1109/ICSE.2015.79

[35] R. Qiu, C. S. Pasareanu, and S. Khurshid, "Certified symbolic execution," in *Automated Technology for Verification and Analysis - 14th International Symposium, ATVA 2016, Chiba, Japan, October 17-20, 2016, Proceedings*, ser. Lecture Notes in Computer Science, C. Artho, A. Legay, and D. Peled, Eds., vol. 9938, 2016, pp. 495–511. [Online]. Available: https://doi.org/10.1007/978-3-319-46520-3_31

[36] R. Qiu, S. Khurshid, C. S. Pasareanu, and G. Yang, "A synergistic approach for distributed symbolic execution using test ranges," in *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017 - Companion Volume*, S. Uchitel, A. Orso, and M. P. Robillard, Eds. IEEE Computer Society, 2017, pp. 130–132. [Online]. Available: https://doi.org/10.1109/ICSE-C.2017.116

[37] T. Xie, N. Tillmann, J. de Halleux, and W. Schulte, "Fitness-guided path exploration in dynamic symbolic execution," in *Dependable Systems & Networks, 2009. DSN'09. IEEE/IFIP International Conference on*. Citeseer, 2009, pp. 359–368.

[38] X. Xiao, S. Li, T. Xie, and N. Tillmann, "Characteristic studies of loop problems for structural test generation via symbolic execution," in *2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013, Silicon Valley, CA, USA, November 11-15, 2013*, E. Denney, T. Bultan, and A. Zeller, Eds. IEEE, 2013, pp. 246–256. [Online]. Available: https://doi.org/10.1109/ASE.2013.6693084

[39] L. Zhang, T. Xie, L. Zhang, N. Tillmann, J. de Halleux, and H. Mei, "Test generation via dynamic symbolic execution for mutation testing," in *26th IEEE International Conference on Software Maintenance (ICSM 2010), September 12-18, 2010, Timisoara, Romania*. IEEE Computer Society, 2010, pp. 1–10. [Online]. Available: https://doi.org/10.1109/ICSM.2010.5609672

[40] T. Xie, N. Tillmann, J. de Halleux, and W. Schulte, "Fitness-guided path exploration in dynamic symbolic execution," in *Proceedings of the 2009 IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2009, Estoril, Lisbon, Portugal, June 29 - July 2, 2009*. IEEE Computer Society, 2009, pp. 359–368. [Online]. Available: https://doi.org/10.1109/DSN.2009.5270315

[41] T. Xie, D. Marinov, W. Schulte, and D. Notkin, "Symstra: A framework for generating object-oriented unit tests using symbolic execution," in *Tools and Algorithms for the Construction and Analysis of Systems, 11th International Conference, TACAS 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings*, ser. Lecture Notes in Computer Science, N. Halbwachs and L. D. Zuck, Eds., vol. 3440. Springer, 2005, pp. 365–381. [Online]. Available: https://doi.org/10.1007/978-3-540-31980-1_24

[42] B. Daniel, T. Gvero, and D. Marinov, "On test repair using symbolic execution," in *Proceedings of the Nineteenth International Symposium on Software Testing and Analysis, ISSTA 2010, Trento, Italy, July 12-16, 2010*, P. Tonella and A. Orso, Eds. ACM, 2010, pp. 207–218. [Online]. Available: http://doi.acm.org/10.1145/1831708.1831734

[43] A. Arcuri and G. Fraser, "On parameter tuning in search based software engineering," in *International Symposium on Search Based Software Engineering*. Springer, 2011, pp. 33–47.

[44] ——, "Parameter tuning or default values? an empirical investigation in search-based software engineering," *Empirical Software Engineering*, vol. 18, no. 3, pp. 594–623, 2013.

[45] F. Gross, G. Fraser, and A. Zeller, "Search-based system testing: high coverage, no false alarms," in *Proceedings of the 2012 International Symposium on Software Testing and Analysis*. ACM, 2012, pp. 67–77.

[46] G. Fraser and A. Arcuri, "The seed is strong: Seeding strategies in search-based software testing," in *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*. IEEE, 2012, pp. 121–130.

[47] J. Malburg and G. Fraser, "Combining search-based and constraint-based testing," in *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*. IEEE Computer Society, 2011, pp. 436–439.

[48] G. Fraser and J. T. de Souza, Eds., *Search Based Software Engineering - 4th International Symposium, SSBSE 2012, Riva del Garda, Italy, September 28-30, 2012. Proceedings*, ser. Lecture Notes in Computer Science, vol. 7515. Springer, 2012. [Online]. Available: https://doi.org/10.1007/978-3-642-33119-0

[49] M. Harman and B. F. Jones, "Search-based software engineering," *Information and software Technology*, vol. 43, no. 14, pp. 833–839, 2001.

[50] J. Clarke, J. J. Dolado, M. Harman, R. Hierons, B. Jones, M. Lumkin, B. Mitchell, S. Mancoridis, K. Rees, M. Roper *et al.*, "Reformulating software engineering as a search problem," *IEE Proceedings-software*, vol. 150, no. 3, pp. 161–175, 2003.

[51] M. Harman and P. McMinn, "A theoretical and empirical study of search-based testing: Local, global, and hybrid search," *IEEE Transactions on Software Engineering*, vol. 36, no. 2, pp. 226–247, 2010.

[52] K. Praditwong, M. Harman, and X. Yao, "Software module clustering as a multi-objective search problem," *IEEE Transactions on Software Engineering*, vol. 37, no. 2, pp. 264–282, 2011.

[53] F. Wu, M. Harman, Y. Jia, and J. Krinke, "HOMI: searching higher order mutants for software improvement," in *Search Based Software Engineering - 8th International Symposium, SSBSE 2016, Raleigh, NC, USA, October 8-10, 2016, Proceedings*, ser. Lecture Notes in Computer Science, F. Sarro and K. Deb, Eds., vol. 9962, 2016, pp. 18–33. [Online]. Available: https://doi.org/10.1007/978-3-319-47106-8_2

[54] P. McMinn, M. Harman, G. Fraser, and G. M. Kapfhammer, "Automated search for good coverage criteria: moving from code coverage to fault coverage through search-based software engineering," in *Proceedings of the 9th International Workshop on Search-Based Software Testing, SBST@ICSE 2016, Austin, Texas, USA, May 14-22, 2016*. ACM, 2016, pp. 43–44. [Online]. Available: http://doi.acm.org/10.1145/2897010.2897013

[55] M. Ó. Cinnéide, I. H. Moghadam, M. Harman, S. Counsell, and L. Tratt, "An experimental search-based approach to cohesion metric evaluation," *Empirical Software Engineering*, vol. 22, no. 1, pp. 292–329, 2017. [Online]. Available: https://doi.org/10.1007/s10664-016-9427-7

[56] H. Jiang, K. Tang, J. Petke, and M. Harman, "Search based software engineering [guest editorial]," *IEEE Comp. Int. Mag.*, vol. 12, no. 2, pp. 23–71, 2017. [Online]. Available: https://doi.org/10.1109/MCI.2017.2670459

[57] G. Fraser and A. Arcuri, "Evolutionary generation of whole test suites," in *2011 11th International Conference on Quality Software*. IEEE, 2011, pp. 31–40.