TensileFuzz: Facilitating Seed Input Generation in Fuzzing via String Constraint Solving

Xuwei Liu
Purdue University
West Lafayette, United States
liu2598@purdue.edu

Wei You*
Renmin University of China
Beijing, China
youwei@ruc.edu.cn

Zhuo Zhang
Purdue University
West Lafayette, United States
zhan3299@purdue.edu

Xiangyu Zhang
Purdue University
West Lafayette, United States
xyzhang@purdue.edu

ABSTRACT

Seed inputs are critical to the performance of mutation based fuzzers. Existing techniques make use of symbolic execution and gradient descent to generate seed inputs. However, these techniques are not particular suitable for input growth (i.e., making input longer and longer), a key step in seed input generation. Symbolic execution models very low level constraints and prefer fix-sized inputs whereas gradient descent only handles cases where path conditions are arithmetic functions of inputs. We observe that growing an input requires considering a number of relations: length, offset, and count, in which a field is the length of another field, the offset of another field, and the count of some pattern in another field, respective. String solver theory is particularly suitable for addressing these relations. We hence propose a novel technique called TensileFuzz, in which we identify input fields and denote them as string variables such that a seed input is the concatenation of these string variables. Additional padding string variables are inserted in between field variables. The aforementioned relations are reverse-engineered and lead to string constraints, solving which instantiates the padding variables and hence grows the input. Our technique also integrates linear regression and gradient descent to ensure the grown inputs satisfy path constraints that lead to path exploration. Our comparison with AFL, and a number of state-of-the-art fuzzers that have similar target applications, including Qsym, Angora, and SLF, shows that TensileFuzz substantially outperforms the others, by 39% - 98% in terms of path coverage.

CCS CONCEPTS

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

KEYWORDS

fuzzing, software testing, dynamic analysis

*Corresponding author.

ISSTA '22, July 18–22, 2022, Virtual, South Korea

ACM ISBN 978-1-4503-9379-9/22/07…$15.00
https://doi.org/10.1145/3533767.3534403

ACM Reference Format:
Xuwei Liu, Wei You, Zhuo Zhang, and Xiangyu Zhang. 2022. TensileFuzz: Facilitating Seed Input Generation in Fuzzing via String Constraint Solving. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '22), July 18–22, 2022, Virtual, South Korea.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3533767.3534403

1 INTRODUCTION

Mutation based fuzzing [1, 53] is an important technique for input generation to expose potential problems in software, which may not have its source code available. Starting from some initial inputs called *seed inputs*, fuzzing techniques mutate pieces of the inputs following certain strategies, for instance, using the coverage improvement as the guidance. The effectiveness of fuzzing techniques hinges on the quality of seed inputs [44, 52]. A seed input driving program execution to interesting components is critical as input mutation can focus on exploring the neighbouring input space to expose defects in these components. However, if high quality seed inputs are not available, mutation fuzzing becomes challenging due to the difficulty of deriving such inputs from scratch. This paper aims to improve seed input generation in mutation based fuzzing when both seeds and source codes are not available.

There are various existing techniques that can be leveraged to address the problem. Symbolic/concolic execution [11, 12, 15, 21, 39] models individual program paths to symbolic constraints, resolving which produces inputs to follow those paths. They can generate inputs from scratch. While these techniques are often sound and feature accuracy in constraint construction, they entail heavy-weight per-instruction modeling and some of the constraints are expensive to solve or even undecidable by their nature. In addition, since the constraints are very low level, some simple relations (such as length of a data field) are not explicitly modeled but rather implicitly convoluted with many other constraints, making resolution difficult (see Section 2 for an example). There are a number of proposals to combine symbolic execution with mutation fuzzing [48, 54, 55]. These techniques alternate between symbolic execution and fuzzing such that symbolic execution helps penetrate when fuzzing fails to make progress. Recently, there are techniques that leverage gradient descent based multi-objective search to generate (seed) inputs, such as Angora [13], its extended version Matryoshka [14], and SLF [52]. These techniques avoid heavy-weight per-instruction modeling in symbolic execution. Instead, they use simpler analysis to derive the gradients of program path conditions regarding input elements. These gradients represent how much change can be induced at path conditions while the input elements are mutated. Inputs are hence mutated based on the gradients.

From our perspective, input generation is essentially search over an intrinsically constrained input space. That is, individual fields in the input have (complex) correlations. Various kinds of existing techniques have different methods of modeling and respecting such constraints. Vanilla fuzzing has the least modeling of the constraints. Hence it is light-weight, scalable, but often less effective (than others). Gradient descent based techniques consider only the mathematical relations and leave the others unmodeled and eventually handled by random fuzzing. Grammar based fuzzing/testing [23, 24] models syntactic constraints as grammar rules. Symbolic execution provides the most general modeling of constraints, by analyzing individual statements and path conditions. It is low level and supports modeling all kinds of constraints as long as they manifest themselves through program statements. However, such manifestation is often too low-level. In many cases, the constraints critical for input growth (e.g., input length constraint) do not explicitly manifest themselves through program statements. Instead, they are implicitly convoluted with a large number of functional constraints that determine the computation to perform, making them unnecessarily difficult to resolve (see an example in Section 2).

In this paper, we propose a novel technique to explicitly model constraints that are critical to input growth. It does not require heavy-weight program analysis. We observe many growth oriented relations across input fields have similar nature to string constraints. For example, an input field being the offset of another input field (in the whole input) can be precisely modeled by the *indexof* constraint in a string solver. Despite the importance of these relations, they are known to be difficult for symbolic execution based techniques [46, 52] due to the implicit and convoluted modeling. We propose to explicitly model these relations as string constraints, without the need of symbolic execution. In particular, we first use a field probing technique piggybacking on AFL [1], the most popular mutation-based fuzzer, to identify individual fields in the input (see Section 3.2). Multiple consecutive bytes form a field if mutating these bytes have identical effect on coverage (e.g., causing the same input validation failure). Each field is then represented by a string variable. The whole input is hence a concatenation of these string variables. Additional padding string variables are introduced in between field string variables. Instantiating these padding variables (through constraint solving) allows the input to grow. Critical cross-field (string) constraints including *length*, *offset*, and *count* are explicitly derived by observing file I/O changes induced by field changes (see Section 3.3). To achieve code coverage, additional cross-field linear constraints are explicitly derived from path conditions by linear regression on sampled input field values and path condition expression values (see Section 3.4). The linear constraints and string constraints are resolved by an SMT solver. For constraints related to non-linear components and other program behaviors that cannot be precisely derived through mutation such as array-indexing, we do not explicitly model them, but resort to an external gradient descent procedure (see Section 3.5) and random mutation.

Our contributions are summarized in the following.

- We develop a novel seed input generation technique for mutation fuzzing. Starting from a 4-byte empty input, the technique can automatically grow the input while respecting the intrinsic constraints across input fields.

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
0x00000000	50	4b	03	04	00	00	00	00	00	00	00	00	00	00	b7	ef
0x00000010	dc	83	01	00	00	00	01	00	00	00	01	00	04	00	31	31
0x00000020	00	00	00	31	50	4b	03	04	00	00	00	00	00	00	00	00
0x00000030	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
0x00000040	00	00	50	4b	01	02	00	00	00	00	00	00	00	00	00	00
0x00000050	00	00	b7	ef	dc	83	01	00	00	00	01	00	00	00	01	00
0x00000060	04	00	01	00	00	00	00	00	00	00	00	00	00	00	00	00
0x00000070	31	31	00	00	00	31	50	4b	01	02	00	00	00	00	00	00
0x00000080	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
0x00000090	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
0x000000a0	24	00	00	00	50	4b	05	06	00	00	00	00	02	00	02	00
0x000000b0	62	00	00	00	42	00	00	01	00	31						

Figure 1: A valid zip archive file.

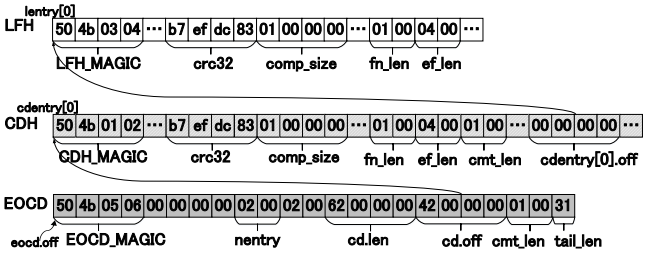

Figure 2: Zip archive structure.

- We propose the novel idea of treating binary fields as string variables and explicitly modeling important cross-field constraints as string constraints.
- In addition to string constraints, we propose to use linear regression to model the linear relations between input fields and path conditions.
- A layered technique is developed, in which (1) an SMT solver is first used to efficiently solve the important but relatively simpler string and linear constraints; (2) gradient descent is used to address non-linear mathematical relations; (3) random mutation is the last resort.
- We develop a prototype TensileFuzz and evaluate it on 12 real-world applications and the Google fuzzer-test-suite benchmark. We compare it with AFL and a number of state-of-the-art fuzzers, including Qsym [54] that combines fuzzing and symbolic execution, Angora [13] and SLF [52] that are gradient descent based. Our results show 39%-98% more coverage. TensileFuzz works on stripped executables and will be publicly available at [3].

Scope. Since our technique relies on AFL's functionalities of mutating inputs and collecting code coverage to construct constraints, it inherits AFL's incapabilities of dealing with programs that require table-based parsing (such as those have LR input grammar like *gcc* and *sqlite*). This is because the code coverage of those programs discloses little about input constraints. Handling those programs is beyond the scope of our paper. Most existing fuzzing techniques, even symbolic execution techniques, have similar limitations unless they are provided with the explicit input grammar [4, 50].

2 MOTIVATION

Zip file format is one of the most popular cross-platform archive formats and is widely used as part of other popular formats such as

```
01 buf = read_from_file();
02 file_len = length_of_file();
03 while(memcmp(buf + eocd.off, EOCD_MAGIC, 4) != 0)
04   eocd.off++;
05 if(eocd.off > file_len) err();                    ⓐ
06 cd.nentry = read_buffer16(buf[eocd.off + 8]);
07 cd.ientry = read_buffer16(buf[eocd.off + 10]);
08 if(cd.ientry != cd.nentry) err();                 ⓑ
09 if(cd.nentry < MIN_ENTRY_REQUIRED) err();         ⓒ
10 cd.len = read_buffer32(buf[eocd.off + 12]);
11 cd.off = read_buffer32(buf[eocd.off + 16]);
12 if(cd.off + cd.len > eocd.off) err();             ⓓ
13 eocd.cmt_len = read_buffer16(buf[eocd.off + 20]);
14 tail_len = buffer_left_len(buf[eocd.off + 22]);
15 if(eocd.cmt_len != tail_len) err();               ⓔ
16 left = cd.len, i = 0, off = 0;
17 while(left > 0)
18   if(left < CDENTRYSIZE) err();                   ⓕ
19   if(memcmp(buf[cd.off+off],CDH_MAGIC,4))err();   ⓖ
20   fn_len = read_buffer16(buf[cd.off + off + 28]);
21   cde_len = CDENTRYSIZE + fn_len;
22   cdentry[i] = readcd(buf[cd.off + off], cde_len);
23   left -= cde_len, off += cde_len;
24   if(++i == cd.nentry && left != 0) err();        ⓗ
25 for(i = 0; i < cd.nentry; i++)
26   j = cdentry[i].offset + cdentry[i].comp_size
27     + cdentry.fn_len + RECORDSIZE;
28   if(j > cd.offset) err();                        ⓘ
29   lentry[i] = readl(buf[cdentry[i].off], j);
30   check_cons(cdentry[i], lentry[i]);
31   crc32 = compute_crc32(lentry[i].data);
32   if(crc32 != lentry[i].crc32) err();             ⓙ
```

Figure 3: Critical validation checks of libzip.

DOCX, EPUB and JAR. A valid zip archive as in Figure 1 may contain several compressed or uncompressed files. Its format is shown in Figure 2, which consists of three sections: the first row *local file header* (LFH), the second row *central directory header* (CDH) and the third row *end of central directory* (EOCD). EOCD contains the entry number, that is, the number of files in the archive, followed by the length of CDH and its offset in the file. CDH is located in the middle of a zip archive and is divided into multiple central directory entries. Each central directory entry contains the offset of the corresponding local entry (in LFH) and additional entry information, including modification date, size, name, etc. LFH is at the beginning and made of local entries which store the file content as well as the entry information like that in CDH. When libzip is used to read a zip archive, it has to perform a sequence of validity checks that distribute in multiple functions of the library. We summarize these checks in Figure 3 and briefly explain them.

Libzip first checks the validity of the file, by searching for the magic number of EOCD at lines 3-4 in Figure 3. In the input example in Figure 1, the magic number is at offset *0xa4-0xa7*. This allows locating the EOCD structure. It then reads the entry number at offset *0xac-0xad* at lines 6-7 and checks the two copies of entry number are identical (line 8) and both larger than 1 (line 9). It identifies the CDH length at *0xb0-0xb3* (line 10) and the CDH offset at offset *0xb4-0xb7* (line 11) from EOCD. The check at line 12 ensures that there is no overlap between CDH and EOCD sections. When reading CDH, it traverses every central directory entry in order (lines 16-24) and finds the corresponding local entries in LFH (lines 25-32). While reading each central directory entry, libzip first checks the remaining length of CDH section is at least as large as a central directory entry (line 18) and the to-be-read entry starts with a magic number (line 19). The central directory entry information is stored in the `cdentry[]` array. When reading LHF, according to `cdentry[i].off` (*0x6c-0x6f* and *0xa0-0xa4*) found in CDH, libzip accesses every local entry in LFH and checks the consistency of the central directory entry and the corresponding local entry (line 30). It also continuously checks if the LFH section overlaps with the CDH section (line 28). Finally libzip returns file data in each local entry if its CRC32 (at offset *0x0e-0x11*) in the entry matches the one calculated from the data (line 32). The input file has to pass all the aforementioned checks. Failing any of them leads to immediate termination with an error message.

Symbolic Execution. Symbolic execution engines (e.g. KLEE [11] and S2E [15]) execute programs symbolically and derive symbolic expressions for variables during execution. When branches are encountered, they fork states to explore both paths and invoke SMT solver to generate test cases satisfying the corresponding symbolic path constraints. For our example, symbolic execution can easily handle simple numeric checks such as checks ⓑ and ⓒ that are difficult for vanilla fuzzers. However, symbolic execution relies solely on the program statements along the path-to-explore to derive constraints. Such constraints are usually very low level, highly convoluted, and incapable of expressing *growth related constraints* in many cases. For example, the input field `cd.off` (read at line 11) denotes the offset of CDH and field `cd.len` denotes the length of CDH. However, such simple offset and length relations do not have any corresponding explicit symbolic constraints. Instead, they are implicitly denoted by a (large) number of low level constraints that are convoluted with other functionalities. Assume in an invalid input, `cd.len` is smaller than the length of CDH, which is very likely as we start with a simple 4-byte input. Such invalidity cannot be directly detected. Instead, symbolic execution can pass check ⓓ that inspects CDH does not overlap with EOCD. At line 16, the invalid `cd.len` value is passed to variable `left` and used in the loop in lines 17-24, which reads in individual central directory entries and reduces `left`. The invalidity will eventually be detected at ⓕ after the symbolic execution engine unrolls the loop for a number of times such that the remaining bytes are insufficient for an entry. Many symbolic execution engines have path exploration strategies that prioritize code coverage. Since unrolling a loop iteration usually does not bring new code coverage, it has a low priority. That is to say, the invalidity of the input will not be detected until very late. To make the situation worse, a symbolic execution engine is not aware of that the root cause of the problem is the mismatch of the length field and the corresponding CDH. Instead, it picks a path condition to negate, hoping to achieve new code coverage. In this case, negating any path condition does not lead to a direct fix of the problem. The engine ends up being stuck in the loop at lines 17-24 and fails to grow the input. Furthermore, symbolic execution engines require specifying input length to begin with. Without knowing the precise input format, providing a proper length is difficult.

Hybrid Fuzzing. Hybrid fuzzing (e.g. Driller [48] and Qsym [54]) combines fuzzing and symbolic execution. They often start with mutation fuzzing and resort to symbolic execution when encountering difficult checks. They hence have the advantages from both

sides, such as applicability and practicality from fuzzing and precise modeling and constraint solving from symbolic execution to penetrate difficult checks. Despite their great potential, it remains challenging to determine when to switch between the two as it is hard to predict when the fuzzer is truly stuck. Furthermore, its capabilities to penetrate difficult checks are bounded by symbolic execution. In our example, hybrid techniques still have difficulty with check ⓕ due to the reasons mentioned before.

Gradient based Fuzzing. Gradient based fuzzers (e.g. Angora [13] and SLF [52]) leverage gradient to guide fuzzing. These fuzzers group input bytes into fields by taint analysis or probing, and then use fields as the basic unit in fuzzing. Instead of solving precise symbolic constraints, they change input fields related to a target predicate along the direction indicated by gradients. For our example, Angora can pass checks ⓐ, ⓑ but fails at check ⓒ due to the *gradient disappearance problem*, namely, changing a field in check ⓒ results in the failure of a previously passed check ⓑ which makes ⓒ not even reachable. SLF leverages a multi-goal gradient based search algorithm to solve inter-dependent numerical checks. However, it still has substantial difficulties on length and offset related checks. It simply degrades to AFL when encountering such checks. It hence cannot penetrate checks ⓕ, ⓗ and ⓘ.

Our Technique. Instead of constructing constraints by modeling individual program statements like in symbolic execution, we derive a set of constraints critical to input growth by probing, i.e., mutating inputs and observing the corresponding execution changes. These constraints include the length, offset and count relations across fields. Reasoning about them entails varying input length and is hence difficult for symbolic execution, which usually requires fixed-size input. We observe that string solver [29] is particularly suitable for reasoning about such constraints and string solvers [28, 56] can shrink, extend, and even shuffle strings, which are the type of mutations we are looking for. We hence propose to consider an input byte sequence as a string such that the aforementioned constraints can be modeled with string operators such as *concat*, *length*, and *indexof*. In the reminder of the paper, we use the term input string to denote a sequence of input bytes. Besides explicitly modeling string constraints, we use linear regression over sample values collected through multiple mutation executions to derive linear correlations cross fields. These constraints are explicitly resolved by an SMT solver. To handle nonlinear arithmetic relations, we leverage gradients. Specifically, when SMT fails to produce an input to follow the path we want, we further change the input fields that have non-linear relations with path conditions following their gradients. With the new valuations of non-linear input fields (generated by gradient descent), we alternate to SMT constraint solving. The procedure is iterative and alternative. The generated seed inputs are then passed to AFL for random mutation.

For the motivation example, TensileFuzz starts with a 4-byte empty input and it can automatically grow the length through probing, string constraint solving and gradient descent solving. It correctly identifies various offset, length and count relations, e.g., between `cd.len` and CDH, between `cd.offset` and CDH, and between `cd.nentry` and (the number of entries in) LHF. As such, the string constraints ensure each generated seed input is structurally valid. It generates the first seed for libzip in 30 minutes and generates 19 valid seeds in 24 hours which lead to 642 explored paths. Here, a valid seed is one that passes all the input checks in Figure 3. In contrast, mutation fuzzer AFL, hybrid fuzzer Qsym, and gradient descent based fuzzer Angora fail to generate any valid seed within 24 hours. SLF generates only 6 seeds.

3 DESIGN

3.1 Overview

We illustrate the workflow of TensileFuzz in Figure 4. In Step A, TensileFuzz starts with an input and mutates every byte of the input to generate many new inputs and execute the target program with these inputs. In Step B, it tracks paths of these inputs and acquires the branches affected by different bytes. It then groups consecutive bytes that affect the same branches into fields. In Step C, string variables are introduced to denote these fields such that an input is a concatenation of these variables. To construct string constraints representing length, offset, and count relations across fields, additional string variables are introduced to denote the possible padding between any two fields. For instance, deciding that a field denotes the length of another data field can be achieved by observing the correlation between changing the value of the field and the need of instantiating the padding field right after the raw data field (e.g., increasing the length field value by 1 entails adding a one-byte padding after the data field). String constraints alone are not sufficient to facilitate program path exploration. Therefore in Step D, our technique additionally constructs a set of path constraints that are integrated with the string constraints. Different from existing symbolic/concolic execution techniques that require modeling each instruction to symbolic constraint, our technique constructs path constraints through probing and linear regression. Specifically, by mutating fields and observing the predicates that are affected, it identifies the set of input fields that should be involved in the path constraints denoting those predicates. Linear regression is used to derive a linear approximation of each path constraint. To handle non-linear arithmetic relations (check ⓙ in Figure 3) that lead to UNSAT of the string and linear constraints, in Step E, gradient descent is used to generate concrete valuation for those fields involved in the non-linear relations, hoping the SMT solver can produce a SAT solution. A SAT solution from the solver is validated to see if it indeed leads to the intended path. If so, it is a new seed input and added to the queue. If not, which means the SAT input leads to a path that deviates from the intended one due to inaccuracy in constraint construction, the invalid input goes through steps A~D to refine the constraints. Seed inputs in the queue are synchronized with AFL for further random mutation. AFL may produce new seed inputs as well. They are synchronized back to TensileFuzz to enhance our seed generation.

3.2 Field Probing

Constraints are present in between fields instead of individual bytes. As such, the first step is to identify fields in the input string. We leverage a field probing technique proposed in [52]. We briefly discuss it for the sake of completeness. Specifically, field probing executes the target program on an input and then leverage the fuzzing engine to observe the lhs and rhs values of each program predicate. It mutates each byte of the input to generate a list of mutated inputs, each having exactly one byte mutated. It feeds the

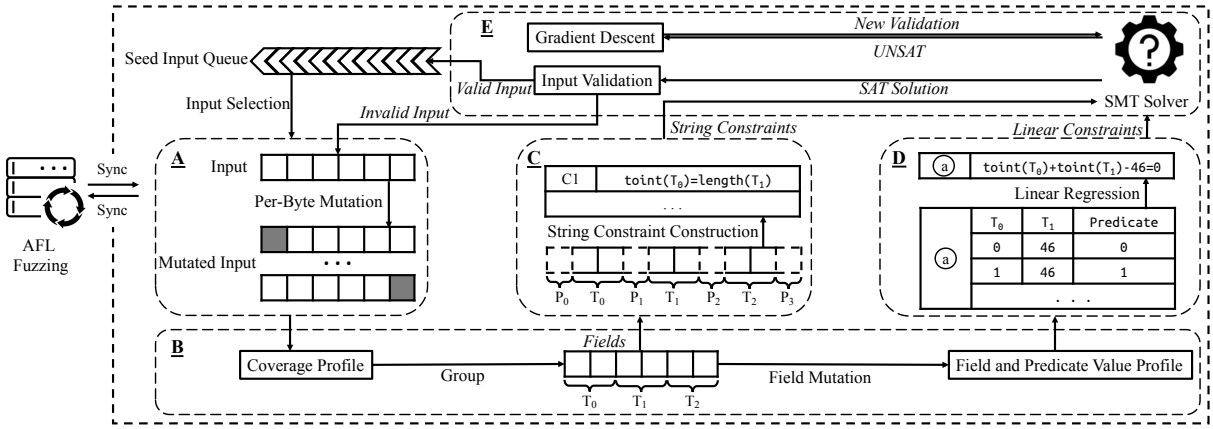

Figure 4: Overview of TensileFuzz.

$$
\begin{array}{lll}
\langle \textit{Term:bool} \rangle & ::= & (Var : bool) \\
& | & true \\
& | & false \\
\langle \textit{Term:int} \rangle & ::= & (Var : int) \\
& | & ConstInt \\
& | & (T_0 : int)\{+, -, *, /\}(T_1 : int) \\
& | & length((T_0 : string)) \\
& | & cnt((T_0 : string), (T_1 : string)) \\
& | & indexof((T_0 : string), (T_1 : string)) \\
& | & toint((T_0 : string)) \\
\langle \textit{Term:string} \rangle & ::= & (Var : string) \\
& | & ConstString \\
& | & concat((T_0 : string), (T_1 : string)) \\
\langle \textit{Expr:bool} \rangle & ::= & (T_0 : bool) \\
& | & (T_0 : bool) = (T_1 : bool) \\
& | & (T_0 : int) \ \ \{=, \geq, >, \leq\} \ \ (T_1 : int) \\
& | & (T_0 : string) = (T_1 : string) \\
& | & not(Expr : bool) \\
& | & (Expr : bool) \wedge (Expr : bool) \\
\langle \textit{Assertion} \rangle & ::= & assert(Expr : bool)
\end{array}
$$

Figure 5: Constraint syntax.

program with the mutated inputs and observes the rhs/lhs changes at individual predicates. The consecutive bytes that induce changes on a same set of predicates are grouped to a field.

3.3 String Constraint Construction

As mentioned in Section 1, the key feature of our technique is to explicitly model the input growth relations as string constraints such that the theory of string solver can be leveraged. Otherwise, they are implicitly encoded as the convolution of a large number of path conditions (as illustrated in Section 2) and become very difficult to satisfy. As shown in Figure 5, we support multiple string operations: *length*, *offset*, and *cnt*, *toint*, and *concat*, representing the length of a string, the offset of a sub-string in another string, and the number of occurrences of some pattern denoted by a string in another longer string, the integer value denoted by a string, and the concatenation of a list of given strings, respectively.

Here, we call the fields probed by the technique mentioned in the last section the *primitive fields* and their concatenation *composite field*. After field identification, we introduce a string variable to denote each primitive field. The overall input string S is hence the concatenation of all these string variables. In addition, we introduce a set of so called *padding* string variables in between primitive fields and at the beginning/end of the input to denote the possible expansion of the input. Recall that we generate seed inputs by gradually disclosing more constraints and expanding the input string (starting from the original four bytes). Assume we have identified n primitive fields, denoted by T_0, T_1, ..., and T_{n-1}. We denote the whole input string as S and the padding strings as P_i We have the following *global composition constraint.*

$$S = concat(P_0, T_0, P_1, T_1, \ldots, T_{n-1}, P_n) \tag{1}$$

We also have a list of *primitive length constraints* that dictate the length of individual primitive fields equals to the probed length. Let l_i be the length of the ith primitive field. We have the following.

$$l_i = length(T_i) \tag{2}$$

Note that in contrast padding variables do not have primitive length constraints. While the above constraints can be directly derived from field probing results, the key challenge lies in identifying the *length*, *offset*, and *cnt* relations across fields.

Deriving Length Constraints Across Fields. These constraints are in the following form.

$$toint(T_0) = length(T_1) \tag{3}$$

It denotes that T_0 is the length of T_1. In our motivation example in Figure 1, we have the following for the CDH field ($0x42 - 0xa3$) and its length field `cd.len` at ($0xb0 - 0xb3$).

$$toint(T_{0xb0-0xb3}) = length(T_{0x42-0xa3})$$

To detect such relations, we intercept all the file read operations. Without losing generality, we assume all these operations take a position parameter denoting the starting position of the read, and a bytes-to-read parameter. We mutate each primitive field and observe if any mutation leads to change of the bytes-to-read parameter at some file read API. If so, a length relation is detected. Specifically, for each field T_i, we mutate it such that $toint(T_i)$ is increased by 1 and 2, generating two mutated inputs. We then execute the program with the mutated inputs. If we observe at some file read API whose original starting position is denoted as s and original

bytes-to-read is denoted as l, the bytes-to-read changes to $l^{[+1]}$ and $l^{[+2]}$ in the mutated executions, we introduce the following constraint, in which T_j, T_{j+1}, ... and T_{j+m} denote the list of fields for the bytes in between s and $s+l$.

$$toint(T_i) \times (l^{[+1]} - l) = length(concat(P_j, T_j, P_{j+1}, T_{j+1}, \ldots, T_{j+m}, P_{j+m+1})) \iff l^{[+1]} - l \equiv l^{[+2]} - l^{[+1]} \quad (4)$$

Intuitively, by increasing the length field by 1 and 2 and observing the corresponding bytes-to-read parameter changes, we identify the unit of the length field. When such unit is consistent (as implied by the condition $l^{[+1]} - l \equiv l^{[+2]} - l^{[+1]}$), we have the constraint in Formula 4. Note that the presence of the padding variables allows the string solver to expand the data field while respecting the length constraint. In other words, expansion is achieved by valuating some padding variable(s) to non-empty string(s). For example, as shown in Box (I) in Figure 6, the 4-bytes field at offset $0xb0$ is changed from $0x62$ to $0x63$ in red (and then $0x64$, which is omitted from the example). The table below presents the intercepted parameters of a file read at $pc1$. Observe that the starting position $s = 0x42$ and the bytes-to-read parameter changes from $l = 0x62$ to $l^{[+1]} = 0x63$ (in red). This allows us to derive the first constraint below box (I).

Deriving Offset Constraints Across Fields. These constraints are in the following form.

$$toint(T_0) = indexof(T_1, T_2) \quad (5)$$

It means that T_0 represents the offset of T_2 in T_1, with T_1 and T_2 usually composite fields. In our motivation example in Figure 1, we have the following for the CDH field ($0x42 - 0xa3$) and its offset field `cd.offset` at $0xb4 - 0xb7$.

$$toint(T_{0xb4-0xb7}) = indexof(S, T_{0x42-0xa3})$$

Similar to length constraints, the derivation of offset constraints leverages the file read interface. Note that an offset field may not be relative to the beginning of the whole input string (but rather the beginning of some internal structure) and its unit may not be byte. As such, the value of an offset field may not be equivalent to the starting position parameter s in the file read. However, it can be easily inferred that s is a linear function of the offset field T_0, as shown in the following.

$$s = a \times toint(T_0) + b \quad (6)$$

Coefficients a and b denote the unit of the offset and the starting position of the internal structure regarding which offset is represented. They can be resolved by observing s changes while mutating T_0. Similar to the derivation of length constraints, for each field T_i, we mutate it such that $toint(T_i)$ is increased by 1 and 2, generating two mutated inputs. We then execute the program with the mutated inputs. If we observe at some file read API whose original starting position is denoted as s and original bytes-to-read is denoted as l, the starting position changes to $s^{[+1]}$ and $s^{[+2]}$, we introduce the following constraint, in which T_j, T_{j+1}, ... and T_{j+m} denote the list of fields for the bytes in between s and $s+l$.

$$toint(T_i) \times (s^{[+1]} - s) = indexof(S, concat(P_j, T_j, P_{j+1}, T_{j+1}, \ldots, T_{j+m}, P_{j+m+1})) \iff s^{[+1]} - s \equiv s^{[+2]} - s^{[+1]} \quad (7)$$

The condition of the constraint represents the sanity check that it is a legitimate offset field, conforming to the above linear function.

For example, as shown in Box (II) in Figure 6, the 4-bytes field at offset $0xb4$ (in black) is changed from $0x42$ to $0x43$ in red (and then $0x44$, which is omitted from the example). The table below presents the change of intercepted parameters of the file read at $pc1$. Observe that originally the starting position $s = 0x42$ and the bytes-to-read parameter $l = 0x62$. The starting position changes to $s^{[+1]} = 0x43$ in red (and then $s^{[+2]} = 0x44$). This allows us to derive the second constraint below box (II).

Deriving Count Constraints Across Fields. These constraints are in the following form.

$$toint(T_0) = cnt(T_1, T_2) \quad (8)$$

It means that T_0 is the number of occurrences of a structural pattern denoted by T_2 in T_1. T_2 is a prefix of T_1. We call T_2 the *pattern field* and T_1 the *data field*. In our motivating example in Section 2, the field `cd.nentry` ($0xae - 0xaf$) denotes the number of central directory entries (e.g., the sequence from $0x42 - 0xa4$) in CDH. Hence, we have the following.

$$toint(T_{0xae-0xaf}) = cnt(T_{0x42-0xa4}, T_{0x42-0x75})$$

Count constraints share similarity with length constraints as the latter also denotes the number of elements in a data field. The difference is that in length constraints, the elements in the data field have a uniform size (e.g., byte). In contrast, in count constraints, the elements have homogeneous structure but various sizes. For example, the entries in CDH, one for each file, may be of different sizes, e.g., depending on the length of the file name represented by the entry. Count constraints are identified through the file read operations as well. However, instead of leveraging the parameters of file reads, we make use of the number of file reads. That is, changing a count field affects the file reads related to the corresponding data field. The number of file reads y is hence a linear function of $toint(T_{cnt})$ if T_{cnt} is a count field.

$$y = a \times toint(T_{cnt}) + b \quad (9)$$

Here, a denotes the number of file reads dedicated to an element in the data field (e.g., an entry in CDH) and b the number of file reads unrelated to the data field (e.g., all the reads unrelated to entries in CDH). Specifically, for each field T_i, we mutate it such that $toint(T_i)$ becomes 0 and 1, generating two respective mutated inputs. We then execute the program with the inputs and count the number of file read operations. Assume the original number of file reads, and the numbers for mutated inputs are y, $y^{[0]}$ and $y^{[1]}$, respectively. We introduce the following constraint, in which T_j, T_{j+1}, ... and T_{j+m} denote the list of fields for the bytes read in the execution of y but not in that of $y^{[0]}$, T_j, T_{j+1}, ... and T_{j+r} the list of fields for the bytes read in the execution of $y^{[1]}$ but not in that of $y^{[0]}$ (i.e., the first element representing the structural pattern), and v_i the original value of $toint(T_i)$.

$$toint(T_i) = cnt(concat(P_j, T_j, P_{j+1}, T_{j+1}, \ldots, T_{j+m}, P_{j+m+1}), concat(P_j, T_j, P_{j+1}, T_{j+1}, \ldots, T_{j+r})) \iff (y - y^{[0]}) \equiv (y^{[1]} - y^{[0]}) \times v_i \quad (10)$$

The condition of the constraint represents the sanity check to make sure T_i is a legitimate count field (conforming to the earlier linear function).

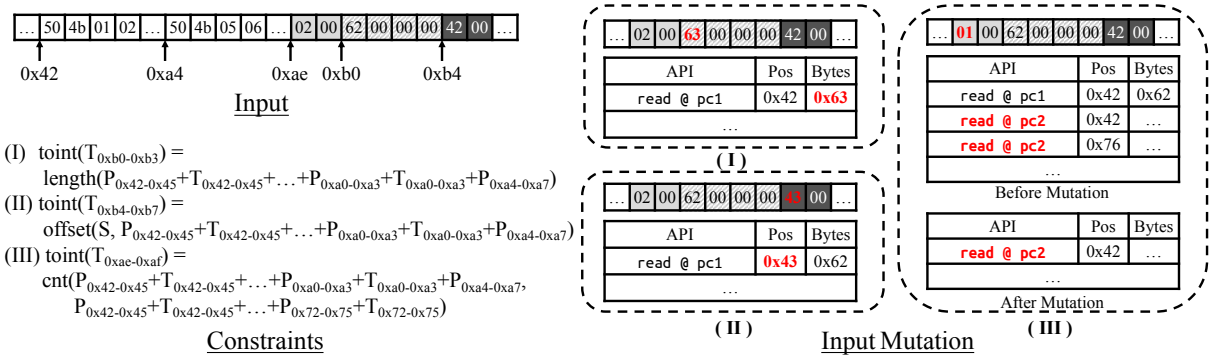

Figure 6: String constraint generation.

For example in box (III) in Figure 6, the 2-byte field at offset $0xae$ (in green) is changed from $0x02$ to $0x01$ (in red). Note that according to the tables before (above) and after (below) the mutation, the occurrences of the file read at $pc2$ is reduced from 2 to 1, which gives rise to the constraint in the bottom of box (III), the composite pattern string is from $0x42-0x75$ as indicated by the pos parameters of the reads at $pc2$.

Solving Count Constraints. Different from *length*, *offset*, and *toint* operations, *count* operation is not natively supported by existing string solvers (e.g., [56]). Since seed inputs are usually small, we bound the count such that we can use an axiom to unfold a count constraint to a set of natively supported constraints. Specifically, the axiom transforms Constraint 10 to the conjunction of the following Constraints 11 and 12.

$$toint(T_i) < 3 \tag{11}$$

$$\begin{array}{ll} (toint(T_i) == 1 \wedge m == r) \vee & [1] \\ (toint(T_i) == 2 \wedge m == 2r) \vee & [2] \\ ((toint(T_i) == 2 \wedge m == r) \wedge & \\ P_{j+m+1} = concat(P'_j, T'_j, P'_{j+1}, T'_{j+1}, \ldots, T'_{j+m}) \wedge & \\ constraint\_clone(P_j, \ldots, T_{j+m}, P'_j, \ldots, T'_{j+m}) & [3] \end{array} \tag{12}$$

Specifically, Constraint 11 decides the unroll bound. For simplicity of illustration, we assume all counts are smaller than 3. Constraint 12 is a disjunction of three possible cases. In the first two cases, as denoted by the subformula [1] and [2], the number of primitive fields in the (composite) data field and the (composite) pattern field matches the count field T_i. In the third case, denoted by subformula [3], the count field has value of 2 while the data field contains only one instance. Therefore, we assert that P_{j+m+1} has the same composition and inner length, offset constraints as the pattern field. The function $constraint\_clone()$ copies all these constraints internal to P_j, T_j, ..., T_{j+m} to the corresponding P'_j, T'_j, ..., T'_{j+m}. For example, if we have $toint(T_j) = length(concat(T_{j+1}, P_{j+1}))$, we have $toint(T'_j) = length(concat(T'_{j+1}, P'_{j+1}))$. Intuitively, subformula [3] allows the input to grow by instantiating P_{j+m+1} to a string that is isomorphic to the pattern string. The transformation for a larger bound can be similarly derived.

3.4 Constraints from Program Paths

The string constraints in the previous section represent those related to input growth. There are constraints for other aspects. In this section, we discuss how we generate these constraints from program paths, without the heavy-weight instruction level tracing or modeling (like taint analysis). The full constraint syntax is presented in Figure 5. Observe that in addition to string operations, we support *bool* and *int* operations. We will discuss how to handle other types of program behaviors without constructing explicit constraints in Section 3.5.

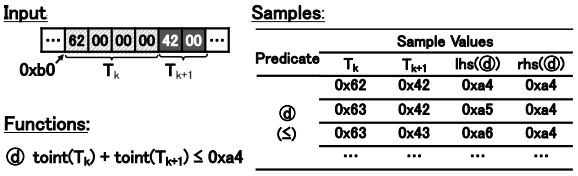

Figure 7: Path constraint generation.

In order to achieve the goal, we follow a standard path exploration method [11, 37], in which a new path is acquired by negating a selected predicate of an existing path. We select the predicates that are input related (as reported in the field probing phase). For a path that we aim to explore, we identify the primitive fields involved in each predicate along the path. This can be done by mutating fields and observing the lhs and rhs value changes for the predicates. In most cases, the lhs/rhs of predicates are merely linear functions of the fields. As such, we use linear regression on the observed lhs/rhs values and the corresponding field values (through multiple mutation runs) to infer such relations and construct linear constraints, which can be resolved by the integer theory in an SMT solver.

Deriving Linear Constraints. For each input-related predicate p along a path (to explore), without losing generality, we assume there are a list of fields related to the lhs expression of p, whose value is denoted as $lhs(p)$. The list may not be complete. We will discuss the effect of incompleteness later this section. A naive approach would be to speculate $lhs(p)$ has a linear relation with the correlated fields and then use linear regression to recover the precise linear form. However in practice, it is possible that $lhs(p)$ has non-linear relation with some of the fields, which would fail the naive linear regression. Inspired by concolic execution, our strategy is to construct explicit constraints only for the linear part of the correlations and assert the input fields that induce non-linear correlations to be fixed.

```
1  x = y = get_input_int();
2  if (x < 10) y = 3 * x;
3  if (y < 20) error();
```

Figure 8: Example for unsoundness.

Specifically, for each T_i related to p, we mutate it a few times and observe the changes of $lhs(p)$. If the changes are linear, we can derive the slope a_i of the linear relation for T_i by $a_i = \frac{lhs'(p)-lhs(p)}{toint(T_i')-toint(T_i)}$. Assume the subset of linearly correlated fields are T_{i1}, T_{i2}, ..., and T_{im}. We derive the following linear constraint.

$$lhs(p) = a_{i1}toint(T_{i1}) + a_{i2}toint(T_{i2}) + \cdots + a_{im}toint(T_{im}) + b \quad (13)$$

The coefficient b is the aggregation of all the contributions from the fields having non-linear correlations. It can be determined through linear regression. In addition, for each input field T_j with value v_j and non-linear correlation with $lhs(p)$, we assert its value stay the same, in the following form.

$$toint(T_j) = v_j \quad (14)$$

Similar constraints can be derived for the $rhs(p)$. The $lhs(p)$ and $rhs(p)$ can then be connected through the comparative operator to constitute the constraint for the predicate.

Example. Figure 7 shows an example for linear constraint generation for check ⓓ. Two fields T_k and T_{k+1}, which correspond to the bytes shown on the top, are related to $lhs(ⓓ)$. Recall ⓓ is a check for non-overlapping EOCD and CDH. The table presents the mutated field values and the corresponding *lhs* and *rhs* values of the predicate. Our analysis derives the following constraint.

$$lhs(ⓓ) = toint(T_{0xb0-b3}) + toint(T_{0xb4-b5})$$

Similarly, we have $rhs(ⓓ) = 0xa4$. If we aim to explore the false branch of ⓓ, we have $lhs(ⓓ) <= rhs(ⓓ)$.

Soundness and Completeness. For the sake of scalability, our technique does not resort to heavy-weight program analysis. As a result, our constraint construction is neither sound nor complete. It is unsound because the input-predicate relation is derived by probing (a kind of sampling), which may not disclose the true relation. Consider the code in Figure 8. Assume x has the value of 6. With a few additional mutations, e.g., $x = 7$ and $x = 8$, our technique derives a linear constraint for line 3: $3 * x >= 20$ if the false branch is intended. However, the constraint is unsound as y is a piece-wise function of x, depending on the condition at line 3. And as we are not explicitly modeling some program behaviors such as symbolic array indexing and high-order functions, the constraint construction is incomplete. The consequences of unsoundness and incompleteness include that (1) a solution to the derived constraints may not drive program execution to follow the intended path; (2) constraints being UNSAT does not mean there does not exist a valid seed input. However, we argue that a full-fledged, sound and complete constraint modeling is often too expensive and leads to constraints very difficult to solve. Our technique, on the other hand, features low cost and capabilities of modeling relations that are fundamental (e.g., related to input growth).

3.5 Iterative Constraint Solving

We resort to an external iterative procedure to mitigate the consequences of incompleteness and unsoundness. In particular, for a new path to explore, we construct a set of constraints as mentioned earlier and pass them to the SMT solver. If there is a solution, we execute the program with the solution and validate that the intended path is taken by the input. If so, the solution is a valid seed input and added to the seed input set. Otherwise, it is likely due to the incompleteness and unsoundness of our technique. In this case, we will redo the field probing and constraint derivation with the new input. Note that since the new input deviates from the intended path, it likely discloses the incorrect/missing relations from the previous round of constraint derivation.

Example. Recall the example that illustrates the unsoundness of constraint derivation in Figure 8. With the current input value $x = 6$, our technique derives constraint $3 * x >= 20$ for line 3 when the false branch is intended. Assume the solver returns $x = 15$ as the solution. Executing the program with the input causes line 2 to take the false branch. As a result, $y = x$ instead of $y = 3x$ and hence the true branch of line 3 is taken instead of the intended false branch. In other words, the solution $x = 15$ is not valid. In this case, our technique will probe and construct the constraints again. It hence derives $x >= 20$ as the constraint for line 3 in order to take the false branch. The solver returns a valid solution $x = 25$. □ Reprobing and reconstruction continues until a valid solution is produced or there is no new relations to be discovered, which is usually due to non-linear relations are not modeled. In this case, we leverage a multi-goal gradient descent method like that in [14, 52] to generate new values for input fields that have non-linear relations with predicate expressions. It is external to the aforementioned iterative SMT constraint solving procedure. The generated new values are explicitly integrated into our SMT constraints through assertions and then we restart the iterative SMT constraint solving.

4 EVALUATION

4.1 Experiment Setup

Our evaluation is performed on two datasets. One contains 12 real world applications from binutils, Google fuzzer-test-suite [2] and other commonly fuzzed programs in other projects [14, 52, 53] for comparison purpose. The other includes 15 programs from the Google fuzzer-test-suite. Each of these programs has a number of planted targets. The way to use the suite is to report how many these targets can be reached by a fuzzer. The two sets cover various kinds of applications, including image, audio, compression, executable, document and font format. We configure fuzzing without source code or *any valid seeds* and we compare TensileFuzz with 4 popular/state-of-the-art fuzzing tools which do not require source code. For vanilla fuzzers, we compare with the baseline AFL which is the most popular fuzzing tool. For hybrid fuzzers combining fuzzing with concolic execution, we compare with Qsym. For gradient guided fuzzers, we compare with Angora and SLF which mitigate inter-dependent constraints by multi-goal-gradient based search. While there are many other fuzzers, as far as we know, these black box fuzzers have the state-of-the-art reported results (e.g., SLF outperforms KLEE and Qsym outperforms Driller [48] and Vuzzer [43]). We do not compare with data-driven fuzzers [47, 49],

Table 1: Evaluation for real world application

Program	AFL		Qsym		Angora		SLF		TensileFuzz	
	Path	Seed	Path	Seed	Path	Seed	Path	Seed	Path	Seed
libzip	172	0	316	0	337	0	381	6	642	19
exiv2	1230	843	1886	1316	1468	826	1413	1101	2105	1958
giflib	167	0	180	17	228	16	276	26	328	53
libpng	375	0	777	0	527	0	674	0	1046	1
libtiff	1011	62	977	69	1060	78	1173	112	1613	114
openjpeg	853	113	860	663	783	537	858	577	988	652
lame	1269	1073	1614	1512	2114	1976	1320	1172	1372	1260
odt2txt	243	0	420	0	279	0	342	0	637	0
7z	1035	1	1123	2	/	/	1307	4	1519	8
readelf	4119	3662	5138	4600	5857	1056	5131	4521	6758	5826
objdump	650	193	1372	371	1451	369	981	324	1627	897
size	1060	323	1159	432	1460	514	1215	479	1587	659

as those fuzzers need a large training corpus (of seeds) that is not available in our setting. We do not compare with fuzzers that target on applications requiring table-based parsing [7, 50], as these applications are beyond our scope.

All tools run along with AFL in their parallel fuzzing modes for best performance. All of our experiments are performed on a machine with 12 cores (Intel® Core™ i7-8700 CPU @ 3.20GHz) and 16GB memory running Ubuntu 16.04 operating system.

4.2 Real World Applications

We run TensileFuzz and other tools on 12 real world programs starting with a 4-byte empty seed. We follow the configurations recommended in a recent fuzzer evaluation work [30] and run each experiment for 24 hours to observe the change of path coverage. We also count the total number of valid seeds for which programs return 0 to figure out the ability of generating structurally valid seeds that pass validation checks. All results are the median of 5 runs and statistically significant with maximum p values less than 0.05. They are presented in Table 1 and Figure 9, where X-axis represents time and Y-axis represents the number of covered paths for each experiment. Angora does not work for 7z as it fails to perform taint analysis.

From the table and the figure, we can make the following observations. Firstly, it is not surprising to see that AFL does not work well without valid seeds. At the very beginning, AFL can efficiently mutate input to cover environment related branches and some shallow validation checks so the coverage increases like other tools. Then it quickly gets stuck after reaching the first difficult check. In most cases, it struggles passing the check in the remaining time and does not make any progress. For example, AFL gets stuck within 2 hours when fuzzing odt2txt and consequently leads to only 2/5 path coverage of TensileFuzz.

Secondly, Qsym helps AFL to penetrate some difficult checks by concolic execution and hence achieves better path coverage. However, since it does not explicitly model constraints related to input growth, it generates fewer seeds than TensileFuzz leading to less path coverage in most cases.

Table 2: Growth related constraint identification. TP denotes True Positive, FP denotes False Positive. A., S., and T. denotes AFL, SLF, and TensileFuzz, respectively.

Program	Total	TP			FP			Program	Total	TP			FP		
		A.	S.	T.	A.	S.	T.			A.	S.	T.	A.	S.	T.
libzip	10	0	10	7	0	23	1	lame	2	0	0	0	0	0	0
exiv2	2	1	0	1	9	0	0	odt2txt	10	1	0	6	0	0	0
giflib	1	0	0	1	2	0	0	7z	6	1	0	4	0	0	2
libpng	2	2	0	2	0	0	0	readelf	11	1	0	4	8	2	0
libtiff	2	1	0	1	6	0	0	objdump	11	1	0	4	16	0	0
openjpeg	3	2	0	3	2	0	0	size	11	1	0	4	16	0	0

Table 3: Contribution breakdown. SLS, GDS, and ICS denote String & Linear Solving, Gradient Descent Solving, and Iterative Constraint Solving, respectively.

Program	SLS	GDS	ICS	Program	SLS	GDS	ICS
libzip	88.98%	4.72%	6.30%	lame	21.31%	67.21%	11.48%
exiv2	84.55%	10.91%	4.55%	odt2txt	90.85%	5.99%	3.17%
giflib	54.76%	35.71%	9.52%	7z	63.87%	27.99%	8.14%
libpng	93.48%	6.52%	0.00%	readelf	75.28%	22.47%	2.25%
libtiff	63.76%	29.53%	6.71%	objdump	76.67%	23.09%	0.23%
openjpeg	26.09%	53.62%	20.29%	size	79.11%	20.67%	0.22%
Overall	SLS: 68.23%			GDS: 25.70%			ICS: 6.06%

Thirdly, gradient based fuzzers rely on gradients and are efficient in solving arithmetic relations such as CRC checks, allowing to achieve reasonable path coverage. Angora achieves much better results in lame than other programs because there are lots of arithmetic relations in lame. However, in some cases, inter-dependent checks may prevent them from exploring deeper paths because the gradients disappear when they go deep. For example, Angora quickly gets stuck in libzip and makes no progress for a long time. SLF leverages multi-goal gradient search to mitigate the inter-dependency so that it can generate valid seeds for such cases. However, both SLF and Angora lack the ability to grow the input efficiently.

TensileFuzz can generate valid seeds for all applications except odt2txt, which requires xml parsing. These seeds are structurally correct and helps pass most validation checks. As such, it yields the best path coverage for all the applications except lame. The average improvements over AFL, Qsym, Angora, SLF are 98%, 39%, 44%, 40%, respectively. From Figure 9, TensileFuzz reaches good coverage in a much faster pace in most cases (due to its ability of growing input properly). TensileFuzz does not work well on lame, which is an mp3 player, because mp3 format has many bit level growth related constraints that are not currently handled.

Growth Related Constraint Identification. We measure the accuracy of string constraint identification of TensileFuzz and compare with AFL and SLF. These constraints, including length, offset and count are growth related. AFL and SLF have type reverse engineering capabilities that can identify fields. Such information may be used to construct constraints. The ground truth is acquired by manually checking how applications handle each byte. Table 2

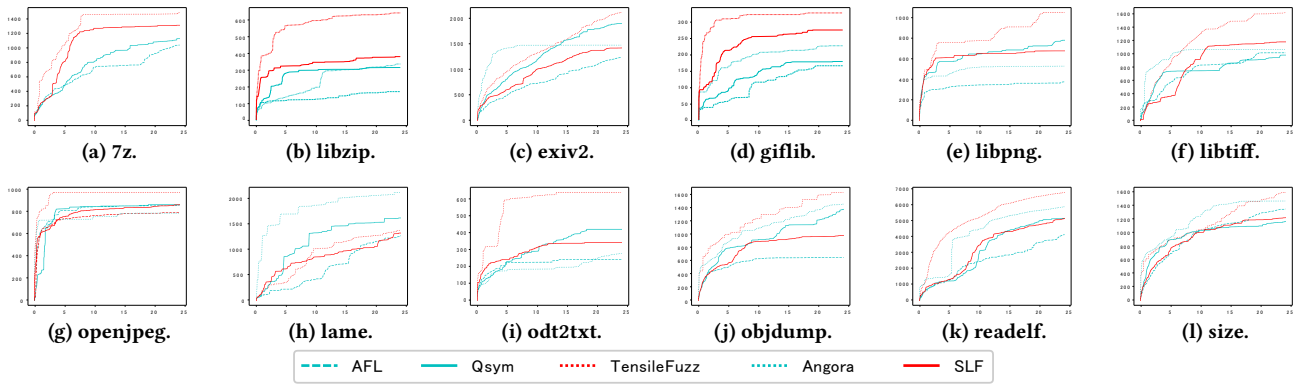

Figure 9: Path coverage. X-axis: time (bounded at 24 hours), Y-axis: the number of unique paths.

represents the result. Total means the number of growth related relations in a valid seed. TP and FP show how many fields are correctly or mistakenly identified, respectively. Observe that our tool has the highest accuracy, explaining why it generates more structurally correct seeds and thus a higher coverage. It misses some cases due to various reasons, including the inherent uncertainty in probing (file read API changes not due to field perturbations) and lack of support of bit fields.

Contribution Breakdown. In this experiment, we measure the contributions of individual solving components in satisfying constraint queries, each query consisting of a set of constraints denoting the intention of exploring some selected branch while respecting the growth related conditions. Table 3 shows the results, each column presents the percentage of queries solved by each component. Observe on average string & linear solving contributes the most (68.23%), followed by gradient descent solving (25.70%) and then iterative constraint solving (6.06%). For some applications (e.g., openjpeg, lame), the iterative constraint solving plays an important role. The reason is that these applications contain a reasonable number of inter-dependent predicates, which might cause field probing fail in string & linear solving, and the gradient disappearance in gradient descent solving. Iterative constraint solving compensates the unsoundness and incompleteness of the other two solving components to some extent.

Case study. We use openjpeg as a case study. Figure 10 shows parts of its validation checks on a BMP file. The program first reads offset and length related fields and perform some checks on these fields. Then it tries to read some bytes at the position specified by the offset. Finally it checks whether the bitcount is 24. Since we start from a four byte seed, the length of the seed is usually small. Many fuzzers got stuck at line 4 because they cannot read enough bytes. Symbolic execution cannot help in this situation because it first tries to change the length to be read to pass the check. However, some other checks for those length related fields may fail afterwards. If the file length is not large enough, the only way to pass line 3 is to make the offset very small, which is of low priority for symbolic execution. Hence, hybrid fuzzers rely on random mutation to grow the file length and then pass this check. However, our tool explicitly denotes such constraints. It recognizes

```
1  (offset,width,height,bitcount) = get_input_ints();
2  length = width * height * bitcount / 8;
3  if (read( length, offset ) != length) error();
4  if (width<MIN_WIDTH || height<MIN_HEIGHT) error();
5  if (bitcount != 24) error();
```

Figure 10: Checks for openjpeg.

the width, height and bitcount fields are related to the length of a structure which starts at the given offset so that it passes line 3. When it further tries to pass lines 4 and 5, the string constraints allow it to stretch the file and grow its length.

4.3 Google Fuzzer Test Suite

Google fuzzer-test-suite includes 24 programs across various categories. Eight of them rely on table-based parsers, which are out of scope and wpantund is not tagged with any buggy locations and excluded. Therefore, we choose the remaining 15 programs. Although directed fuzzers[8, 31] is known to be effective in reaching buggy lines, they require source code to do annotation so we do not compare with them. Note that we always start with a 4-byte empty seed, making it very difficult to reach the buggy locations. In fact, none of the tools can reach any locations for 9 out of the 15 programs. Hence we only present the results of 6 programs that at least one tool has reached some buggy location(s). In total, they correspond to 14 different buggy locations. Table 4 shows the results. It shows the programs, the tagged location (i.e., target) in the program and the time for each tool to reach the location (in minutes).

From the table, we can see AFL reaches 6 locations within 24 hours while Qsym and SLF can reach 7 locations. Our tool is able to reach 12 locations, including all the locations reached by other tools. Specifically, Qsym is able to pass shallow checks quickly but it spends much more time solving constraints as paths become longer. It reaches 7 targets, a subset of ours, and takes 80% more time to reach these places compared to ours. Angora is much more efficient and has similar time performance as ours but it gets stuck as more path conditions become inter-dependent. It can only reach 4 out of 14 locations, only 1/3 of ours. SLF reaches 7 locations within a reasonable time, but also takes 38% more time than ours.

Table 4: Evaluation for Google Fuzzing Test Suite. T/O denotes tool cannot reach locations in 24 hours. / denotes tool cannot work on the program.

Program	Location	Tool				
		AFL	Qsym	Angora	SLF	TensileFuzz
libjpeg	jdmarker.c:659	1245.26	12.07	/	378.8	403.67
woff2	woff2_dec.cc:500	T/O	T/O	T/O	T/O	900.79
	woff2_dec.cc:1274	45.26	14.80	10.53	37.08	5.35
llvm-libcxxabi	first crash	47.25	46.37	42.00	49.53	48.06
vorbis	codebook.c:479	T/O	T/O	/	T/O	1206.00
	codebook.c:407	T/O	T/O	/	T/O	T/O
	res0.c:690	T/O	T/O	/	T/O	T/O
libpng	png.c:1035	187.23	812.76	0.87	273.00	1.72
	pngread.c:757	T/O	T/O	T/O	37.61	36.27
	pngrutil.c:1393	T/O	836.63	T/O	T/O	473.20
	pngread.c:738	190.65	822.40	19.22	29.40	15.43
	pngrutil.c:3182	T/O	T/O	T/O	T/O	55.47
	pngrutil.c:139	0.25	0.45	0.45	0.33	0.30
libarchive	archive...warc.c:537	T/O	T/O	T/O	T/O	73.96

TensileFuzz is most efficient and outperforms others both in time and the number of reached locations.

We also investigate why none of the tools can reach any locations in the other 9 programs. None of the tools work well on network protocol programs (i.e., libssh, openssl, boringssl and c-ares) because these programs typically have built-in arrays which are difficult for symbolic execution and multiple inter-dependent non-linear predicates which prevents searching by gradients. TensileFuzz degrades to Angora because these programs do not have string constraints and have many inter-dependent non-linear constraints. For the other programs (e.g., lcms, harfbuzz and freetype), they accept many different kinds of inputs. It is rather difficult to reach a specific location without the guidance of diverse seeds.

5 RELATED WORK

Generation-based fuzzing. Generation-based fuzzing generates inputs from file format specification either manually provided or learnt from a large corpus [24, 49]. Some works[10, 19, 25, 35] leverage reverse engineering to recover the file format by control flow or data flow. For example, Mimid[25] takes a program and a small set of sample inputs and automatically infers a readable context-free grammar capturing the input language of the program. It infers the syntactic input structure by observing access of input characters at different locations of the input parser. Tupni[19] can automatically identify record sequences and record types in input formats and find different types of constraints on the values of fields. It can generalize input formats over multiple inputs, including protocols and binary file format. Although these techniques are effective in generating valid seeds, they require initial valid seed corpus for learning purpose. In contrast, our work does not need initial valid seed inputs. Instead, it relies on fuzzing and concolic execution to generate seed inputs and learn constraints on the fly.
Mutation-based fuzzing. Mutation-based fuzzing generates inputs by mutating the prepared seed inputs. There are a large number of projects aiming to improve mutation effectiveness using internal program states [33, 43], statistical metrics [9, 32] and online input probing[53]. Another line of work focuses on hybrid fuzzing, which integrates fuzzing with constraint solving. Path constraints are collected and solved by invoking symbolic execution engines. Driller [48] is the first fuzzer combining concolic execution with fuzzing at run time. It begins with AFL and seeks help from concolic execution when it gets stuck in some hard constraints like magic numbers. Qsym[54] improves the performance of concolic execution by optimizing emulation speed and reducing emulation usage. Intriguer[16] optimizes symbolic execution with field-level knowledge and efficiently performs symbolic emulation for more relevant instructions. REDQUEEN[5] utilize the idea of observing the correspondence between the input fields and program state and invoke smt solver to solve magic bytes and (nested) checksum tests. Other work uses a lightweight gradient descent to solve path constraints. Angora[13] is the first fuzzer which introduces gradient descent to fuzzing. It mutates fields by the gradient computed from predicate values and field values. SLF[52] and Matryoshka[14] improve Angora with multi-goal search algorithms aiming at passing nested constraints. Eclipser[17] uses linear regression to simplify the path constraints and solve them by gradient descent, which is similar to our path constraints solving technique. Technically speaking, TensileFuzz is also a hybrid fuzzing approach, which features a high-level constraint modeling and a layered constraint solving technique.
Symbolic Execution. Symbolic execution provides an automatic mechanism for exploring program paths. Prior work has proposed several optimizations at the levels of the execution engine [41, 42, 51] and the constraint solver [20, 36]. Specific accelerated solving algorithms have been proposed for certain program structures (e.g., arrays [38] and loops [46]). Different from existing constraint optimization methods, TensileFuzz proposes a new way to *explicitly* model input growth related constraints, which are critical in fuzzing. Our technique is also related to string constraint solving [6, 18, 22, 26, 27, 34, 40, 45], by being a client analysis.

6 LIMITATION

TensileFuzz identifies input fields at the byte level by grouping consecutive bytes so it can only work for chunk based binary programs but not text/grammar based programs. In addition, the constraints built by TensileFuzz are usually under-constrained because it models only string constraints and linear path constraints. The missing non-linear constrains are handled by an external gradient descent engine. As a result, the performance of TensileFuzz degrades when programs have many complex bit/non-linear checks.

A few programs may not invoke file I/O APIs for many times. Instead, they read the entire file to a buffer and process the buffer. TensileFuzz can not rely on file read APIs to build string constraints but applies heuristic rules to infer constraints among fields. The string constraints can be more inaccurate in this case. Better heuristic rules help mitigate the problem.

7 CONCLUSION

We develop a seed input generation technique for mutation fuzzing. It features explicitly modeling constraints critical to input growth as string constraints and solving them with a string solver. It additionally models constraints important to path exploration using linear regression and gradient descent. Our experiments show that our technique TensileFuzz is highly effective, out-performing the state-of-the-art that targets the same set of applications.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for their constructive comments. The Purdue authors were supported in part by NSF1901242 and 1910300, ONRN000141712045, N000141410468 and N000141712947. The RUC author was supported in part by NSFC under grants 62002361 and U1836209, and the Fundamental Research Funds for the Central Universities and the Research Funds of Renmin University of China under grant 22XNKJ29. Any opinions, findings, and conclusions in this paper are those of the authors only and do not necessarily reflect the views of the sponsors.

REFERENCES

[1] 2021. American Fuzzy Lop (AFL). http://lcamtuf.coredump.cx/afl.
[2] 2021. Google Fuzzer Test Suite. https://github.com/google/fuzzer-test-suite.
[3] 2022. TensileFuzz. https://github.com/TensileFuzz/TensileFuzz.
[4] Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, Ahmad-Reza Sadeghi, and Daniel Teuchert. 2019. NAUTILUS: Fishing for Deep Bugs with Grammars. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019*. https://doi.org/10.14722/ndss.2019.23412
[5] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. 2019. REDQUEEN: Fuzzing with Input-to-State Correspondence. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. The Internet Society. https://doi.org/10.14722/ndss.2019.23371
[6] Nikolaj Bjørner, Nikolai Tillmann, and Andrei Voronkov. 2009. Path Feasibility Analysis for String-Manipulating Programs. In *Tools and Algorithms for the Construction and Analysis of Systems, 15th International Conference, TACAS 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings*. 307–321. https://doi.org/10.1007/978-3-642-00768-2_27
[7] Tim Blazytko, Cornelius Aschermann, Moritz Schlögel, Ali Abbasi, Sergej Schumilo, Simon Wörner, and Thorsten Holz. 2019. GRIMOIRE: Synthesizing Structure while Fuzzing. In *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*. 1985–2002.
[8] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed Greybox Fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017*. https://doi.org/10.1145/3133956.3134020
[9] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2016. Coverage-Based Greybox Fuzzing as Markov Chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (Vienna, Austria) *(CCS '16)*. Association for Computing Machinery, New York, NY, USA, 1032–1043. https://doi.org/10.1145/2976749.2978428
[10] Juan Caballero, Heng Yin, Zhenkai Liang, and Dawn Song. 2007. Polyglot: Automatic Extraction of Protocol Message Format Using Dynamic Binary Analysis. In *Proceedings of the 14th ACM Conference on Computer and Communications Security* (Alexandria, Virginia, USA) *(CCS '07)*. Association for Computing Machinery, New York, NY, USA, 317–329. https://doi.org/10.1145/1315245.1315286
[11] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. 2008. 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*. 209–224. http://www.usenix.org/events/osdi08/tech/full_papers/cadar/cadar.pdf
[12] Junjie Chen, Wenxiang Hu, Lingming Zhang, Dan Hao, Sarfraz Khurshid, and Lu Zhang. 2018. 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 (LIPIcs, Vol. 109)*, Todd D. Millstein (Ed.). Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 6:1–6:27. https://doi.org/10.4230/LIPIcs.ECOOP.2018.6
[13] Peng Chen and Hao Chen. 2018. Angora: Efficient Fuzzing by Principled Search. In *Proceedings of the 39th IEEE Symposium on Security and Privacy (SP 2018)*. https://doi.org/10.1109/SP.2018.00046
[14] Peng Chen, Jianzhong Liu, and Hao Chen. 2019. Matryoshka: Fuzzing Deeply Nested Branches. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019*. 499–513. https://doi.org/10.1145/3319535.3363225
[15] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. 2011. 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*. 265–278. https://doi.org/10.1145/1950365.1950396
[16] Mingi Cho, Seoyoung Kim, and Taekyoung Kwon. 2019. Intriguer: Field-Level Constraint Solving for Hybrid Fuzzing. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*. 515–530. https://doi.org/10.1145/3319535.3354249
[17] Jaeseung Choi, Joonun Jang, Choongwoo Han, and Sang Kil Cha. 2019. Grey-box Concolic Testing on Binary Code. In *Proceedings of the International Conference on Software Engineering*. 736–747. https://doi.org/10.1109/ICSE.2019.00082
[18] Aske Simon Christensen, Anders Møller, and Michael I. Schwartzbach. 2003. Precise Analysis of String Expressions. In *Static Analysis, 10th International Symposium, SAS 2003, San Diego, CA, USA, June 11-13, 2003, Proceedings*. 1–18. https://doi.org/10.1007/3-540-44898-5_1
[19] Weidong Cui, Marcus Peinado, Karl Chen, Helen J. Wang, and Luis Irun-Briz. 2008. Tupni: Automatic Reverse Engineering of Input Formats. In *Proceedings of the 15th ACM Conference on Computer and Communications Security* (Alexandria, Virginia, USA) *(CCS '08)*. Association for Computing Machinery, New York, NY, USA, 391–402. https://doi.org/10.1145/1455770.1455820
[20] Ikpeme Erete and Alessandro Orso. 2011. Optimizing Constraint Solving to Better Support Symbolic Execution. In *Fourth IEEE International Conference on Software Testing, Verification and Validation, ICST 2012, Berlin, Germany, 21-25 March, 2011, Workshop Proceedings*. 310–315. https://doi.org/10.1109/ICSTW.2011.98
[21] Xi Ge, Kunal Taneja, Tao Xie, and Nikolai Tillmann. 2011. 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*, Richard N. Taylor, Harald C. Gall, and Nenad Medvidovic (Eds.). ACM, 992–994. https://doi.org/10.1145/1985793.1985971
[22] Indradeep Ghosh, Nastaran Shafiei, Guodong Li, and Wei-Fan Chiang. 2013. JST: an automatic test generation tool for industrial Java applications with strings. In *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*. 992–1001. https://doi.org/10.1109/ICSE.2013.6606649
[23] Patrice Godefroid, Adam Kiezun, and Michael Y Levin. 2008. Grammar-based whitebox fuzzing. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 206–215. https://doi.org/10.1145/1375581.1375607
[24] Patrice Godefroid, Hila Peleg, and Rishabh Singh. 2017. Learn&fuzz: Machine learning for input fuzzing. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 50–59. https://doi.org/10.1109/ASE.2017.8115618
[25] Rahul Gopinath, Björn Mathis, and Andreas Zeller. 2020. Mining input grammars from dynamic control flow. In *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*, Prem Devanbu, Myra B. Cohen, and Thomas Zimmermann (Eds.). ACM, 172–183. https://doi.org/10.1145/3368089.3409679
[26] Claudio Gutiérrez. 1998. Solving Equations in Strings: On Makanin's Algorithm. In *LATIN '98: Theoretical Informatics, Third Latin American Symposium, Campinas, Brazil, April, 20-24, 1998, Proceedings*. 358–373. https://doi.org/10.1007/BFb0054336
[27] Pieter Hooimeijer and Westley Weimer. 2010. Solving string constraints lazily. In *ASE 2010, 25th IEEE/ACM International Conference on Automated Software Engineering, Antwerp, Belgium, September 20-24, 2010*. 377–386. https://doi.org/10.1145/1858996.1859080
[28] Adam Kiezun, Vijay Ganesh, Philip J. Guo, Pieter Hooimeijer, and Michael D. Ernst. 2009. HAMPI: a solver for string constraints. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis, ISSTA 2009, Chicago, IL, USA, July 19-23, 2009*. 105–116. https://doi.org/10.1145/1572272.1572286
[29] Adam Kiezun, Philip J. Guo, Pieter Hooimeijer, Michael D. Ernst, and Vijay Ganesh. 2019. Theory and practice of string solvers (invited talk abstract). In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2019, Beijing, China, July 15-19, 2019*. 6–7. https://doi.org/10.1145/3293882.3338993
[30] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating Fuzz Testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang (Eds.). ACM, 2123–2138. https://doi.org/10.1145/3243734.3243804
[31] Gwangmu Lee, Woochul Shim, and Byoungyoung Lee. 2021. Constraint-guided Directed Greybox Fuzzing. In *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*, Michael Bailey and Rachel Greenstadt (Eds.). USENIX Association, 3559–3576. https://www.usenix.org/conference/usenixsecurity21/presentation/lee-gwangmu
[32] Caroline Lemieux and Koushik Sen. 2017. FairFuzz: Targeting Rare Branches to Rapidly Increase Greybox Fuzz Testing Coverage. *CoRR* abs/1709.07101 (2017). https://doi.org/10.1145/3238147.3238176
[33] Yuekang Li, Bihuan Chen, Mahinthan Chandramohan, Shang-Wei Lin, Yang Liu, and Alwen Tiu. 2017. Steelix: program-state based binary fuzzing. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. https://doi.org/10.1145/3106237.3106295
[34] Tianyi Liang, Andrew Reynolds, Cesare Tinelli, Clark W. Barrett, and Morgan Deters. 2014. A DPLL(T) Theory Solver for a Theory of Strings and Regular Expressions. In *Computer Aided Verification - 26th International Conference, CAV*

2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings. 646–662. https://doi.org/10.1007/978-3-319-08867-9_43

[35] Zhiqiang Lin, Xiangyu Zhang, and Dongyan Xu. 2010. Reverse Engineering Input Syntactic Structure from Program Execution and Its Applications. *IEEE Trans. Software Eng.* 36, 5 (2010), 688–703. https://doi.org/10.1109/TSE.2009.54

[36] Hristina Palikareva and Cristian Cadar. 2014. Multi-solver Support in Symbolic Execution. In *Proceedings of the 12th International Workshop on Satisfiability Modulo Theories, SMT 2014, affiliated with the 26th International Conference on Computer Aided Verification (CAV 2014), the 7th International Joint Conference on Automated Reasoning (IJCAR 2014), and the 17th International Conference on Theory and Applications of Satisfiability Testing (SAT 2014), Vienna, Austria, July 17-18, 2014*. 15. https://doi.org/10.1007/978-3-642-39799-8_3

[37] Hui Peng, Yan Shoshitaishvili, and Mathias Payer. 2018. T-Fuzz: fuzzing by program transformation. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 697–710. https://doi.org/10.1109/SP.2018.00056

[38] David Mitchel Perry, Andrea Mattavelli, Xiangyu Zhang, and Cristian Cadar. 2017. Accelerating array constraints in symbolic execution. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, Santa Barbara, CA, USA, July 10 - 14, 2017*. 68–78. https://doi.org/10.1145/3092703.3092728

[39] Suzette Person, Guowei Yang, Neha Rungta, and Sarfraz Khurshid. 2011. Directed incremental symbolic execution. In *Acm Sigplan Notices*, Vol. 46. ACM, 504–515. https://doi.org/10.1145/1993316.1993558

[40] Wojciech Plandowski. 2006. An efficient algorithm for solving word equations. In *Proceedings of the thirty-eighth annual ACM symposium on Theory of computing*. 467–476. https://doi.org/10.1145/1132516.1132584

[41] Rui Qiu, Sarfraz Khurshid, Corina S. Pasareanu, and Guowei Yang. 2017. 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*, Sebastián Uchitel, Alessandro Orso, and Martin P. Robillard (Eds.). IEEE Computer Society, 130–132. https://doi.org/10.1109/ICSE-C.2017.116

[42] Rui Qiu, Corina S. Pasareanu, and Sarfraz Khurshid. 2016. Certified Symbolic Execution. In *Automated Technology for Verification and Analysis - 14th International Symposium, ATVA 2016, Chiba, Japan, October 17-20, 2016, Proceedings (Lecture Notes in Computer Science, Vol. 9938)*, Cyrille Artho, Axel Legay, and Doron Peled (Eds.). 495–511. https://doi.org/10.1007/978-3-319-46520-3_31

[43] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. 2017. VUzzer: Application-aware Evolutionary Fuzzing.. In *NDSS*, Vol. 17. 1–14. https://doi.org/10.14722/ndss.2017.23404

[44] Alexandre Rebert, Sang Kil Cha, Thanassis Avgerinos, Jonathan Foote, David Warren, Gustavo Grieco, and David Brumley. 2014. Optimizing Seed Selection for Fuzzing. In *Proceedings of the 23rd USENIX Security Symposium 2014*.

[45] Prateek Saxena, Devdatta Akhawe, Steve Hanna, Feng Mao, Stephen McCamant, and Dawn Song. 2010. A Symbolic Execution Framework for JavaScript. In *31st IEEE Symposium on Security and Privacy, S&P 2010, 16-19 May 2010, Berleley/Oakland, California, USA*. 513–528. https://doi.org/10.1109/SP.2010.38

[46] Prateek Saxena, Pongsin Poosankam, Stephen McCamant, and Dawn Song. 2009. 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*. 225–236. https://doi.org/10.1145/1572272.1572299

[47] Dongdong She, Kexin Pei, Dave Epstein, Junfeng Yang, Baishakhi Ray, and Suman Jana. 2019. NEUZZ: Efficient Fuzzing with Neural Program Smoothing. In *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*. IEEE, 803–817. https://doi.org/10.1109/SP.2019.00052

[48] 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*, Vol. 16. 1–16. https://doi.org/10.14722/NDSS.2016.23368

[49] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2017. Skyfire: Data-Driven Seed Generation for Fuzzing. In *2017 IEEE Symposium on Security and Privacy, SP 2017*. https://doi.org/10.1109/SP.2017.23

[50] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2019. Superion: grammar-aware greybox fuzzing. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019*. 724–735. https://doi.org/10.1109/ICSE.2019.00081

[51] Tao Xie, Nikolai Tillmann, Jonathan de Halleux, and Wolfram Schulte. 2009. Fitness-guided path exploration in dynamic symbolic execution. In *Dependable Systems & Networks, 2009. DSN'09. IEEE/IFIP International Conference on*. Citeseer, 359–368. https://doi.org/10.1109/DSN.2009.5270315

[52] Wei You, Xuwei Liu, Shiqing Ma, David Mitchel Perry, Xiangyu Zhang, and Bin Liang. 2019. SLF: fuzzing without valid seed inputs. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019*. 712–723. https://doi.org/10.1109/ICSE.2019.00080

[53] Wei You, Xueqiang Wang, Shiqing Ma, Jianjun Huang, Xiangyu Zhang, XiaoFeng Wang, and Bin Liang. 2019. ProFuzzer: On-the-fly Input Type Probing for Better Zero-Day Vulnerability Discovery. In *2019 IEEE Symposium on Security and Privacy, SP 2019*. 769–786. https://doi.org/10.1109/SP.2019.00057

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

[55] Lei Zhao, Yue Duan, Heng Yin, and Jifeng Xuan. 2019. Send Hardest Problems My Way: Probabilistic Path Prioritization for Hybrid Fuzzing. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019*. https://doi.org/10.14722/ndss.2019.23504

[56] Yunhui Zheng, Xiangyu Zhang, and Vijay Ganesh. 2013. Z3-str: a z3-based string solver for web application analysis. In *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26, 2013*. 114–124. https://doi.org/10.1145/2491411.2491456