Intriguer: Field-Level Constraint Solving for Hybrid Fuzzing

Mingi Cho
Yonsei University
imgc@yonsei.ac.kr

Seoyoung Kim
Yonsei University
kseoy4046@yonsei.ac.kr

Taekyoung Kwon
Yonsei University
taekyoung@yonsei.ac.kr

ABSTRACT

Hybrid fuzzing, which combines fuzzing and concolic execution, is promising in light of the recent performance improvements in concolic engines. We have observed that there is room for further improvement: symbolic emulation is still slow, unnecessary constraints dominate solving time, resources are overly allocated, and hard-to-trigger bugs are missed.

To address these problems, we present a new hybrid fuzzer named Intriguer. The key idea of Intriguer is field-level constraint solving, which optimizes symbolic execution with field-level knowledge. Intriguer performs instruction-level taint analysis and records execution traces without data transfer instructions like `mov`. Intriguer then reduces the execution traces for tainted instructions that accessed a wide range of input bytes, and infers input fields to build field transition trees. With these optimizations, Intriguer can efficiently perform symbolic emulation for more relevant instructions and invoke a solver for complicated constraints only.

Our evaluation results indicate that Intriguer outperforms the state-of-the-art fuzzers: Intriguer found all the bugs in the LAVA-M(5h) benchmark dataset for ground truth performance, and also discovered 43 new security bugs in seven real-world programs. We reported the bugs and received 23 new CVEs.

CCS CONCEPTS

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

KEYWORDS

fuzzing; hybrid fuzzing; constraint solving

ACM Reference Format:
Mingi Cho, Seoyoung Kim, and Taekyoung Kwon. 2019. Intriguer: Field-Level Constraint Solving for Hybrid Fuzzing. In *2019 ACM SIGSAC Conference on Computer and Communications Security (CCS '19), November 11–15, 2019, London, United Kingdom.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3319535.3354249

1 INTRODUCTION

Fuzzing is an automated bug-finding technique that iteratively feeds malformed test inputs to a target program [21]. Fuzzers (or fuzzing tools) have become a popular research topic in the computer security community for their successes and challenges: a *coverage-based fuzzer* named AFL [34] has successfully found a great number of bugs in real programs, but it is no good at rendering inputs that lead to deeper execution paths with complex branch conditions. For example, AFL is effective at loose constraints, e.g., `x > 0`, by applying random mutations, but gets stuck when encountering tight constraints like magic bytes, e.g., `x == 0x0f365c01`. On the other hand, symbolic/concolic execution [14, 17] is good at finding exact or narrow range values for tight constraints. Thus, a *hybrid fuzzer* like Driller[29], combining both methods, can take advantage of them; however, this promising approach still suffers from the expensive and slow nature of symbolic emulation in real programs.

Recently, a state-of-the-art approach named Qsym [33] presented a *fast concolic execution engine* to support hybrid fuzzing to mitigate this problem. Qsym implemented instruction-level symbolic emulation integrated with native execution using dynamic binary translation, and leveraged invaluable heuristics, such as basic block pruning and optimistic solving, for performance improvement in test case generation. To assess the improved penetration power, Qsym demonstrated its performance on the benchmark dataset called LAVA-M [11], which is a synthetic test suite that artificially injects *hard-to-reach* bugs into four Linux utilities. Note that many other modern fuzzers [8, 19, 26, 27] also used LAVA-M for benchmarking; and this is significant in terms of assessing the ground truth performance of fuzzing [17].

The result is a bit disappointing, however: Qsym reported that a great number of bugs (42.1% in `who` binary) were still unreachable within the limited time budget (5h per program). It was good at the other three smaller programs, but not good enough at `who`, which contains a larger number of hard-to-reach bugs in LAVA-M. Interestingly, most of the path constraints were *multibyte constraints*, e.g., `0x6c616a93 == (lava_get(3022))`, that would be a serious problem for AFL but should not have been for Qsym. As noted, fuzzing time is a priceless resource for scaling to complicated real programs.

In this paper, (§2) we scrutinize the performance of hybrid fuzzing for hard-to-reach bugs, and (§3) we present a new hybrid fuzzer named *Intriguer* to address the following problems that we observe: (§2.1) symbolic emulation is still slow due to data transfer instructions and input field-agnostics; (§2.2) unnecessary constraints, which can be filtered out, dominate solving time; (§2.3) resources are overly allocated to constraints; (§2.4) hard-to-trigger bugs that are triggered only when arithmetic/branch boundary conditions are simultaneously met, are frequently (or always) missed.

(§3.1) The key idea of Intriguer is *field-level constraint solving*, which optimizes symbolic execution with field level knowledge. (§3.2) Intriguer performs instruction-level dynamic taint analysis (DTA), and records execution traces without data transfer instructions like `mov`. Intriguer further reduces the execution traces (e.g., tainted instructions that accessed a wide range of input bytes) in advance, minimizing symbolic emulation. (§3.3) Intriguer infers

CCS '19, November 11–15, 2019, London, United Kingdom

ACM ISBN 978-1-4503-6747-9/19/11…$15.00
https://doi.org/10.1145/3319535.3354249

fields from unknown input formats, and constructs a field transition tree for each inferred field to lighten symbolic emulation and compose minimized constraints. (§3.4) Thanks to these optimizations, Intriguer significantly reduces symbolic emulation to target relevant instructions only, and runs a constraint solver minimally but effectively. Intriguer uses a symbolic solver for complicated (truly complex) constraints only, and directly solves uncomplicated constraints by the tree. Moreover, Intriguer deals with both branch and arithmetic boundary conditions without loss of generality.

We implement Intriguer and perform experiments to evaluate its effectiveness. To account for the random nature of fuzzing, we refer to the recent guideline of [17]. We directly compare Intriguer with Qsym, VUzzer [27], and AFL in our experiments. We use LAVA-M [11] for ground truth benchmark tests, and real-world programs such as `objdump`, `nm`, `readelf`, `ffmpeg`, `avconv`, `tiff2pdf`, and `bsdtar`.

This paper makes the following contributions:

- **Efficient hybrid fuzzing through field-level approach.** We further improved the performance of hybrid fuzzing by novel techniques like trace reduction and field-level constraint solving based on our thorough analysis. We evaluated how much those techniques improve hybrid fuzzing and how practical it is.
- **Effectiveness to find hard-to-reach and hard-to-trigger bugs.** Intriguer can dig into fast and trigger both types of bugs, e.g., Intriguer found all the bugs in LAVA-M(5h) benchmark programs.
- **Real-world bugs.** Intriguer discovered 43 new bugs in seven real programs, including a new one in `ffmpeg` that has been missed by OSS-Fuzz for four years, and received 23 new CVEs.
- **Open source.** For further research, we release our tool at publication time at https://github.com/seclab-yonsei/intriguer.

Organization: §2 analyzes hybrid fuzzing. §3 and §4 describe Intriguer's design and implementation, respectively. §5 evaluates Intriguer on benchmarks and real programs. §6 discuss limitations and further directions. §7 explains related work. §8 concludes this paper. The appendix (§A~§E) contains supplementary material.

2 MOTIVATION

In this section, we describe the motivation of our study by thoroughly analyzing the performance of hybrid fuzzing, e.g., Qsym. We enumerate four findings and explain the details.

2.1 Symbolic Emulation Is Still Slow

Qsym is an elegant approach of relaxing the performance bottlenecks in hybrid fuzzing by realizing fast, instruction-level concolic execution along with heuristics like optimistic solving and block pruning. Qsym selectively emulates only the instructions necessary to generate symbolic constraints, unlike existing approaches that emulate all instructions in the tainted basic blocks [33].

Table 1 compares the time for emulation and constraint solving, improved by Qsym, with the time for native execution and DTA for a single execution of two LAVA-M programs and three real programs. In programs other than `objdump`, we observe that the emulation time is still significantly slower. Symbolic emulation needs management (generation, modification, and deletion) of symbolic expressions whenever a tainted instruction is executed, resulting in a high overhead. Thus, it is always desirable to further reduce

Table 1: Qsym's symbolic emulation time is compared with native execution, DTA, and constraint solving time for a single execution.

Program	Native (s)	DTA (s)	Qsym	
			Emulation (s)	Solving (s)
md5sum (LAVA)	0.009	4.3	48.0	7.4
who (LAVA)	0.004	3.9	24.6	0.6
ffmpeg	0.009	21.0	501.7	108.3
tiff2pdf	0.001	2.4	493.8	106.2
objdump	0.003	3.1	49.6	550.4

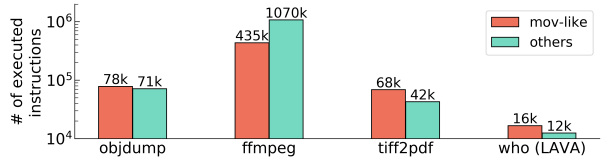


Figure 1: The number of instructions executed by each program: `mov`-like instructions vs. other instructions. (log scale)

Table 2: Qsym's single execution: `objdump` for 600s, default timeout (10s) per constraint. (# total queries: 2413, total solving time: 527s)

Function	# of timeouts	# of queries	Total time (s)
bfd_hash_hash()	14	150	190
bfd_hash_lookup()	2	19	25
bfd_hash_insert()	9	100	209

the emulation time. Figure 1 shows the number of tainted instructions (`mov`-like vs. others). Those instructions will be symbolically emulated unless removed. Interestingly, Qsym stops generating constraints from frequently repeated blocks in its block pruning except for `mov` instructions. Qsym's design decision is understandable because `mov` instructions are used for input data propagation, affecting data flow; however, we observed that they resulted in a significant slowdown in symbolic emulation (§A in appendix).

Our key observation is that we can avoid emulating `mov`-like instructions to further reduce symbolic emulation time because by comparing tainted instructions, before and after `mov`, along with their information (e.g., concrete values) in the inferred field level, we could track where data moves. Unlike other approaches that are field-agnostic, we can optimize symbolic execution with inferred-field knowledge. (§3.4.1)

Our approach. Use light-weight DTA in instruction level to record tainted offsets and concrete values in the execution traces except for `mov`-like instructions, and use the traces for field-level constraint solving. (§3.2, §3.4)

2.2 Unnecessary Constraints Dominate Solving

In Table 1, unlike the rest of the programs, `objdump` took more time for constraint solving than emulation. This indicates that there were a huge number of timeouts (note: we used a default timeout) in using a solver and these situations, requiring repetitive measures, might have caused performance bottlenecks. In basic block pruning, Qsym does not prune constant instructions (e.g., *shifting* and *masking*) or

```
1  struct bfd_hash_entry *
2  bfd_hash_lookup (...)
3  {  ...
4   hash = bfd_hash_hash (string, &len); // compute hash
5   _index = hash % table->size;
6   for (hashp = table->table[_index];
7    hashp != NULL;
8    hashp = hashp->next)
9    {
10    if (hashp->hash == hash  // compare hash
11     && strcmp (hashp->string, string) == 0)
12     return hashp;
13   ...
```

Listing 1: A part of `bfd_hash_lookup()` used in `binutils` for hash lookup. The hash comparison part is not repeatedly executed, but complex constraints are generated by repeated instructions for hashing, making the solving time long.

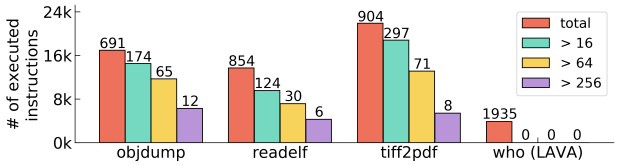


Figure 2: The number of tainted instructions executed by each program according to the input range (bytes). The number above each bar represents the number of unique instructions removing duplicates. Many instructions are repeated over a wide input range.

the `mov` instructions as we already mentioned. These instructions are, however, frequently used in many cryptographic functions, such as hash functions, and apt to generate complex constraints. We observe in `objdump`, that a lot of solving time was needed and repeatedly led to timeout in this context.

Table 2 shows the result of solver timeout (10s), i.e., *unsat*, when Qsym runs `objdump` with an elf format input for 600s timeout. We observe that every branch with a timeout is associated with a hash as can be seen from the function name of `objdump`. Qsym wasted 80.5% of its solving time on complex[1] constraints that are unnecessary, in three hash functions for a single execution of `objdump`. As shown in Listing 1, a hash is computed by an input (line 5) and so accompanies very complex constraints; and therefore, when this hash is compared (line 11) in branch decisions, a timeout occurs.

Our key observation is that the code in which we expect to find bugs is a part of checking the size of data to obtain a hash rather than a part of computing such a hash, or a part that is executed after a hash check; and therefore, it is necessary to lower the priority of the instructions that use a wide range of input offsets (Figure 2).

Our approach. Group the DTA-produced execution traces by instruction, and reduce the traces of the group that accessed a wide range of input bytes. (§3.2).

2.3 Resources Are Overly Allocated

Modern fuzzers treat tight constraints, such as magic bytes, as complex constraints and adopt sophisticated techniques to resolve them. Hybrid fuzzers like Qsym also uses a symbolic solver to deal with such constraints, and other fuzzers like VUzzer and Steelix leverage

[1]At only three branches, 11.1% of total queries were used for those constraints.

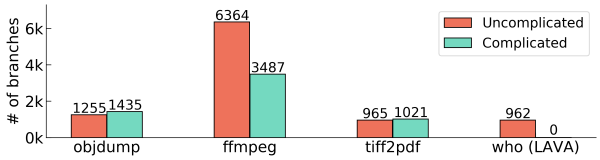


Figure 3: The number of symbolic branches executed. Uncomplicated means a branch that makes a direct comparison with an input, such as loose (a > 0) and tight (b == 0x0f365c01) constraints. Such a great portion of branches can be solved without using a solver.

```
1  int main(int argc, char* argv[]){
2    unsigned int width, height, count, size;
3    unsigned char* data;
4    unsigned char header[4];
5
6    read(stdin, header, 4);
7    ...
8    read(stdin, &width, 4);
9    read(stdin, &height, 4);
10   if(width > 0x5FFF  height > 0x5FFF){
11    error("width or height too large.");
12    exit(1);
13   }
14   count = width * height;
15   size = count * 8;
16   data = (unsigned char*) malloc(size);
17   //read data from standard input
18   for(int i=0; i < count; i++){
19    memset(data, 0, 8);
20    ...
21 }}}
```

Listing 2: An example of hard-to-trigger bugs. State-of-the-art fuzzers have difficulty triggering the bugs even if they execute the buggy code blocks. Table 3 shows our experiment results.

program analysis techniques at the byte level. However, we observe that those constraints are not truly complex, and overly consume resources (§B): if we approach them in field level, it is possible to treat loose constraints (e.g., `x > 0`), multibyte constraints (e.g., `x == 0x0f365c01`), and even narrow range arithmetic constraints (e.g., `x > 0 ∧ x < 10`) in the same way immediately.

In Figure 3, we observe that a great portion of branch conditions are uncomplicated in programs, and they can be directly solved without querying a symbolic solver. In particular, the symbolic branches (962) in LAVA-M `who` binary are actually uncomplicated and they can be immediately solved by instruction-level DTA if input field information is inferred in advance.

Our approach. Classify the complicated constraints and the uncomplicated constraints systematically, and use a symbolic solver for complicated constraints only. (§3.3, §3.4).

2.4 Hard-to-trigger Bugs Are Missed

Listing 2 shows example code for a hard-to-trigger bug, triggered only when exact values causing an integer overflow are inputted[2]. An integer overflow occurs under the condition: `width<=0x5fff∧height<=0x5fff∧width*height*8>0xffffffff`. For instance, for input values `width=0x5557` and `height=0x5fff`, `size` multiplied in line 15 is `0x100025548`, which satisfies the overflow condition,

[2]Note that these bugs were considered in the earlier work of smart fuzzing [23, 31]

Table 3: Results of fuzzing Listing 2 by simulating that the code was exercised by each fuzzer (seed size: 12 bytes). We measured time and number of executions, until triggering the bug for 30 times.

Fuzzer	Runs	Time until trigger		Executions until trigger	
		Median (s)	Mean (s)	Median	Mean
Intriguer	30	2	2	27	27
AFL	30	1,412	2,187	6,612k	10,211k
VUzzer	30	fail (>24h)	fail (>24h)	fail	fail
Qsym	30	fail	fail	fail	fail

going beyond the boundary of `int` and resulting in an integer overflow. The `malloc` function (line 16) allocates memory for `data` of that `size`, leading to a buffer overflow in line 19.

Table 3 shows the results of four fuzzers assumed to have explored the code path. We repeated the experiments for 30 runs per fuzzer, with a 24h time budget for unlimited executions per run. Intriguer took only 27 executions in 2s to trigger the bug in each of 30 runs; however, AFL took much more, and unfortunately Qsym and VUzzer failed in every run. Both `height` and `width` must be in `0x5557~0x5fff` for bug-triggering; however, it is unlikely to coincidentally catch such values in a fuzzing loop. The probability p for the coincident case occurring is: $p = ((\texttt{0x5fff-0x5557+0x1}) / \texttt{0xffffffff})^2 = (6.35 * 10^{-7})^2$. The above example demonstrates that even if the buggy code is executed, an actual bug can be missed if the exact values that trigger it are unknown.

Our approach. Treat the arithmetic boundary conditions like the branch constraints (§3.4).

3 DESIGN

3.1 Overview

Figure 4 illustrates an overview of Intriguer's system architecture as a hybrid fuzzer. The key idea of Intriguer is a field-level constraint solving, which optimizes symbolic execution with field level information. Intriguer takes a target binary program and a test case as input and aims to discover interesting offsets and values to generate new test cases that explore new paths for fuzzing.

To do this, Intriguer performs an instruction-level taint tracking on a target program execution with an input test case provided by a coverage-based fuzzer. Unlike existing approaches, in advance to constraint solving, Intriguer significantly reduces the execution traces: it records the execution traces except for data transfer instructions like `mov`, and removes tainted instructions that accessed a wide range of input bytes (§3.2). Furthermore, Intriguer infers an input field as a chunk of byte offsets, discarding no-field inferred instructions (§3.3.1), and constructs the field transition trees for each inferred field to lighten symbolic emulation and compose minimized constraints (§3.3.3) (§3.4). Thanks to the inferred fields and the trees, Intriguer can invoke an SMT solver for truly complicated constraints only, and directly solve uncomplicated constraints, including tight multi-byte comparisons, for both branch and arithmetic conditions. By doing these, Intriguer can remarkably reduce unnecessary works incurred by symbolic emulation (i.e., to be more relevant to bug-finding) (§3.4.1) and constraint solving (§3.4.2). The rest of this section explains more details about our approaches.

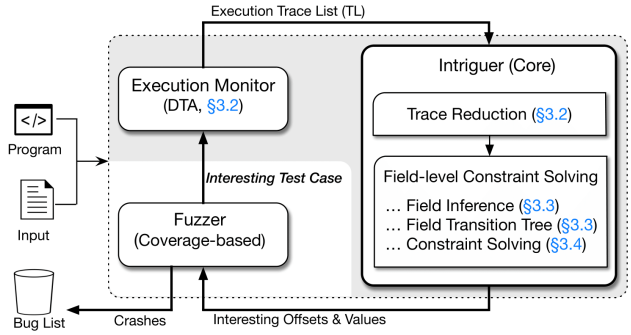


Figure 4: Overview of Intriguer – system architecture.

Table 4: Structure of trace record in CFG format. - means untainted.

<trace>	::=	<address> "." <opcode> "." <operand>
<operand>	::=	"{" <offset> "}{" <value> "}" | "{"<offset>"}{" <value> "}." <operand>
<offset>	::=	"-" | "-," <offset> | <integer> | <integer> "," <offset>
<value>	::=	<integer> | <integer> "," <value>

```
while((c = *s++) != '\0')
{
  hash += c + (c << 17);
  hash ^= hash >> 2;
}
```

```
"0x80c3a07.cmp.{00,-,-,-}{61,00,00,00}.{-,-,-,-}{00,00,00,00}"
"0x80c3a07.cmp.{01,-,-,-}{61,00,00,00}.{-,-,-,-}{00,00,00,00}"
...
"0x80c3a07.cmp.{07,-,-,-}{61,00,00,00}.{-,-,-,-}{00,00,00,00}"
"0x80c3a07.cmp.{08,-,-,-}{61,00,00,00}.{-,-,-,-}{00,00,00,00}"
...
"0x80c3a07.cmp.{FF,-,-,-}{61,00,00,00}.{-,-,-,-}{00,00,00,00}"
```

Figure 5: An example that shows the trace reduction. The left-hand code is a portion of `bfd_hash_hash()` used for hashing in `binutils`. The right-hand trace records (cf. Table 4) show a wide range of input offsets (00~FF) due to `while` loop. Intriguer removes the records except the first eight records (colored in green). We omit 0x for space.

3.2 Execution Trace Reduction

Intriguer performs an instruction-level taint tracking for data flow analysis that enables fine-grained trace reduction and field-level constraint solving. Our basic insight is that existing concolic execution engines waste a great deal of time in symbolic emulation for instructions that are unlikely to be relevant to bug-finding (§5.4). Intriguer reduces the size of the execution trace by removing tainted instructions that are irrelevant, before constraint solving stages.

3.2.1 Execution Trace. The DTA engine of Intriguer (named Execution Monitor) produces an execution trace list (briefly called TL) with the record format specified in Table 4 by tracing a target program execution with an input test case provided by a coverage-based fuzzer. The Execution Monitor records the traces of instructions except for `mov`-like instructions. Note that even if there is no trace of `mov`, it is possible to track where data moves through the traces of the instructions that precede and succeed `mov`, as we mentioned in §2.1. We describe how it works in §3.4.1.

The record of the TL, for each tainted instruction, consists of the instruction's address and opcode, and the operand's input byte offsets and concrete values. An example in Figure 5 shows how the trace record is configured: the *cmp* instruction at memory address 0x80c3a07 repeatedly used two operands, one of which the first byte

Table 5: An Example that shows field inference. Consecutive offsets are inferred as a single field. $F_{(i,j)}$ indexing a starting offset i with size j indicates the consecutive offsets as underlined.

#	Offset	$F_{(i,j)}$	#	Offset	$F_{(i,j)}$
1	{-,-}	-	7	{-,0x00,0x01,-}	$F_{(0,2)}$
2	{-,-,-,-}	-	8	{0x00,0x01,0x02,-}	$F_{(0,3)}$
3	{0x00,-,-,-}	-	9	{0x00,0x01,0x02,0x03}	$F_{(0,4)}$
4	{0x00,-,0x01,-}	-	10	{0x03,0x02,0x01,0x00}	$F_{(0,4)}$
5	{0x00,0x01}	$F_{(0,2)}$	11	{0x00,0x01,10,-}	$F_{(0,2)}$
6	{0x00,0x01,-,-}	$F_{(0,2)}$	12	{0x00,0x01,0x10,0x11}	$F_{(0,2)}$, $F_{(16,2)}$

is tainted by consecutive values of 'a' and the other untainted with values 0x00. Intriguer does not distinguish the operand's type (e.g., register, memory, or immediate), but only considers taint states; and if tainted, the exact input byte offsets, e.g., 0x00~0xFF, are set in the records.

3.2.2 Instruction-level Granularity. Intriguer performs an instruction-level taint tracking, similar to Qsym, but we are more intrigued in instruction-level trace reduction (rather than basic block-level; dissimilar to Qsym) to suppress unnecessary symbolic emulation (Figure 2) and constraint generation caused by redundant instructions (Listing 1) before symbolic execution (§ 5.4).

Figure 5 shows an example in which many instructions (e.g., addition, shift, exclusive or, comparison, and assignment) are repeatedly executed with accessing a wide range of input bytes (e.g., 0x0 ~ 0x1000 in byte offsets) in the `while` loop. In general, however, the location where we expect to find bugs is the part that checks the size of data for hashing, or the part that executes after a hash check, rather than computing hash values. Therefore, we reduce the tainted instructions that accessed a wide range of input bytes among the repetitive instructions. This feature also occurs when performing frequent operations including encryption, decryption, encoding, decoding, compression, and decompression. The trace reduction algorithm is simple and straightforward as described in the next subsection.

3.2.3 Trace Reduction. Intriguer runs the following procedure to produce the reduced trace list (briefly called RL) from the TL.

(1) Group the TL's trace records by "instruction address" and generate a sublist named RL_{addr} for each group.
(2) In each RL_{addr}, count the number of unique offsets, and go to the reduction step if the offset count exceeds a predefined threshold (§5.4).
(3) In the reduction step, leave the first eight trace records only and remove the remaining records in the over-counted RL_{addr}.

An example in Figure 5 shows how this trace reduction procedure simply works. Note that RL is the union of all final RL_{addr}'s where each RL_{addr} is a group of (reduced) records for each tainted instruction. Thanks to grouping in advance, Intriguer can count the number of offsets more efficiently.[3] In the above procedure, Intriguer counts the first offset for each operand and the same offsets only once for each RL_{addr}. The condition of the offset count exceeding the threshold indicates that the corresponding instruction

[3] $O(n^i) \rightarrow O(2n)$ where n and i are the number of trace records and the number of instructions, respectively, in TL.

```
1  static void
2  coff_swap_filehdr_in (bfd * abfd, void * src, void *
       dst)
3  ...
4  /* MS handles overflow of line numbers by carrying
       into the reloc field (it appears).  Since it's
       supposed to be zero for PE *IMAGE* format, that's
        safe.  This is still a bit iffy.  */
5  #ifdef COFF_IMAGE_WITH_PE
6    scnhdr_int->s_nlnno = (H_GET_16 (abfd, scnhdr_ext->
         s_nlnno)
7      + (H_GET_16 (abfd, scnhdr_ext->s_nreloc) << 16));
8    scnhdr_int->s_nreloc = 0;
9  ...
```

Listing 3: An example of multi-field operand in `objdump`.

has accessed a wide range of input bytes (with a number of different offsets beyond the threshold). Note that Intriguer basically uses the threshold value as 16 (§3.2).

One might be concerned that this procedure might remove interesting records. First of all, Intriguer does not remove all traces for the tainted instructions that accessed a wide range of input, but instead it reduces them. Thus, as evaluated in our experiments (§5.4), Intriguer can discover more new test cases and bugs within the same amount of time. Furthermore, Intriguer prioritizes RL but later on it can go back to TL if there is no more new test cases generated. In the fuzzing loop, Intriguer can also gradually increase the threshold when no more new test cases are generated.

3.3 Field Inference and Field Transition Tree

Aiming to discover interesting offsets and values quickly for new test cases, Intriguer infers a field from unknown input formats to directly decide where to mutate in an input test case (§3.3.1), and also to discard no-field inferred instructions. Subsequently, Intriguer constructs a field transition tree to optimize symbolic emulation without needing memory address and register information, and also constraint solving by composing minimized (and relevant) constraints (§3.3.3). These are the essential steps to proceed with a field-level constraint solving to decide what value to mutate (§3.4).

3.3.1 Field Inference. We define a *field* as consecutive input bytes directly used by an instruction. Intriguer infers such a field from unknown input formats[4] by leveraging offset information recorded in the execution trace, e.g., RL. If an operand is tainted by consecutive offsets of the input data in RL, these consecutive offsets can be grouped together and inferred as a single field. For an inferred field, which starts at offset i and has a chunk size j, we denote the inferred field information as $F_{(i,j)}$ and assigns a list of instructions used. Table 5 shows an example of field inference done by Intriguer.

Note that by definition, Intriguer ignores a single byte field for inference. In hybrid fuzzing, the role of a concolic execution engine is to deal with complex constraints; and so Intriguer decides to infer a field for multi-byte offsets only and discard the trace records for a single byte offset. This inference decision makes it possible to conduct further reduction for the execution trace and the following constraint generation; and the fuzzer will happily handle such a single byte constraint. In Table 5, #3 is a single byte offset and #4 is

[4] Similar techniques are popularly used for protocol/file format reverse engineering [5, 9, 10, 32]. Particularly, our method is close to that of Tupni [10], but Intriguer uniquely infers a field for both single and multi-field operands. (cf. #12 in Table 5)

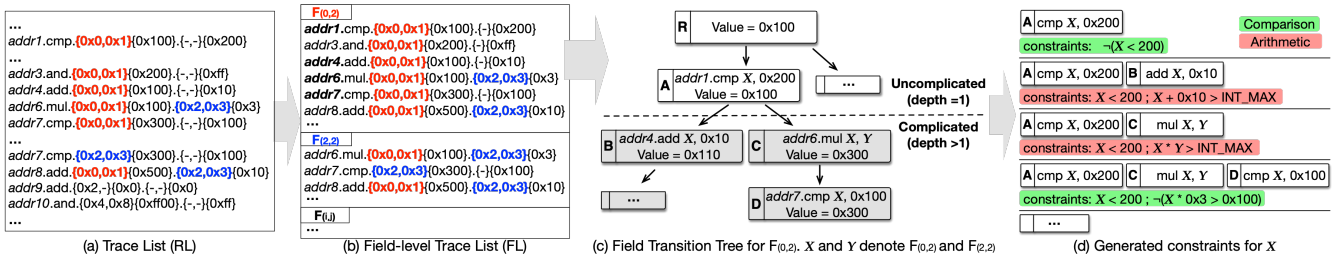


Figure 6: An example of field-level constraint solving. (a) RL is reduced from TL. (b) FL is derived from RL for inferred fields, $F_{(0,2)}$ and $F_{(2,2)}$. (c) A field transition tree is constructed from FL for $F_{(0,2)}$. If depth = 1, uncomplicated constraints are generated. If depth > 1, complicated constraints are generated. (d) On a node with a comparison instruction, a constraint is generated to explore a new branch (green); and on a node with an arithmetic instruction, a constraint that takes a value beyond an integer boundary (red) is generated (e.g., >INT_MAX).

non-consecutive; and they are ignored. #11 shows an example that consecutive two-byte offsets are inferred but a single byte offset is ignored.

In real-world programs, multiple fields are occasionally combined together as an operand (e.g., {0x00,0x01} and {0x10,0x11} as {0x00,0x01,0x10,0x11} in #12, Table 5), which we call a *multi-field operand*. In this case, the field is unlikely inferred by the existing techniques because the trace with a multi-field operand indicates that such offsets are seemingly not consecutive. If such a field is associated with a bug, a false negative could occur; and so the trace of an multi-field operand should be inferred. In this context, Intriguer extends the field inference technique to be able to infer a combination of multiple consecutive offsets, e.g., $F_{(0,2)}$ and $F_{(16,2)}$, from a multi-field operand. Listing 3 shows a real-world example of the multi-field operand, which uses two fields combined into one operand when processing PE files in `objdump`. In line 6, `scnhdr_ext→s_nlnno` and `scnhdr_ext→s_nreloc`, which were read from a file, are assigned to the lower two bytes and upper two bytes of `scnhdr_int→s_lnno`, respectively. As a result, two fields are assigned to `scnhdr_ext→s_nlnno`.

3.3.2 Field-level Trace List. Recall that the execution trace TL was significantly reduced to RL, and the RL was used for field inference. Now Intriguer further groups the trace records of RL according to the inferred fields, and discards all irrelevant (i.e., no-field-inferred) trace records. We call the reconfigured list of such groups a field-level trace list (FL). Obviously, a subset of FL, which is a group for $F_{(i,j)}$, holds trace records in which at least one operand used $F_{(i,j)}$. Figure 6-(b) shows an example of FL derived from Figure 6-(a) according to field inference. Trace record *addr9* of Figure 6-(a) is discarded whereas *addr6* to *addr8* are duplicated in FL. A trace record could belong to multiple groups if distinct fields are used by multiple operands or a multi-field operand is used.

3.3.3 Field Transition Tree. Intriguer builds a field transition tree from FL for each inferred field. The goal of constructing the field transition trees is to optimize symbolic emulation and constraint solving. We explain more details after describing the procedure to build the field transition tree as follows:

(1) Set a root node as a source input value
(2) Set a new node to hold ⟨*trace T*, *value V* (a result of the instruction in the *trace*)⟩ where the value is reproduced through the precedent nodes
(3) Add the new node only if the tainted value and the reproduced value are the same.

Figure 6-(c) shows an example of the field transition tree generated from FL for $F_{(0,2)}$. Intriguer creates the tree by setting a root node (tagged R) with V = 0x100 that is the source input value as shown in $F_{(0,2)}$, and then appends the trace records in FL to the root node in order to complete the tree. In doing so, Intriguer first sets a new node (tagged A) with the instruction (`cmp` at *addr1*) and the operands with regard to the first trace record. Because the first operand of the first trace record uses the field $F_{(0,2)}$, Intriguer searches the tree for the first operand's value 0x100. In the current tree, only the root node exists; and since the root node's V is 0x100, this trace record is appended as a child node to the root node. Next, to add a second trace record, Intriguer searches for a node whose V is 0x200 (first operand's value). However, since there is no node with V 0x200 in the tree, this trace is considered *unreproducible* and is not added to the tree. In this way, Intriguer adds trace records by comparing a *field value* of an operand with V of the node. If there are multiple nodes with the same V, Intriguer selects the most recently appended node. Our insight behind this tree construction is that a tainted operand value is reproducible from a source input value through the instructions properly recorded by the execution monitor.

In concolic execution, similar to DTA, it is necessary to tag the memory addresses and registers affected by the input data propagated by running instructions. Unlike the taint tag that stores only a simple taint state, however, concolic execution needs a *symbolic tag* that contains a symbolic expression representing the memory addresses and registers. In order to deal with the exact symbolic expressions according to the input data flow, it is further necessary to obtain the exact information about the memory addresses and registers whenever running the instructions that perform a write operation onto the memory or register, i.e., not just data transfer instructions but all instructions that incur tag propagation such as `add` and `sub`. Existing concolic executors, both on-line (e.g., Qsym [33]) and off-line (e.g., SAGE [14]), need to generate and manage the symbolic expressions for all tainted memory and registers, incurring a large performance overhead.

Intriguer significantly reduces this overhead by constructing the field transition tree: the tree records the changes in the concrete values of each inferred field, rather than deals with symbolic tags for memory addresses and registers. Our basic insight is that the data flow can be inferred by analyzing the concrete values stored

```
1  int a, b, c;
2  read(0, (int*)&a, 4);   // a is inferred as F(0,4)
3  read(0, (int*)&b, 4);   // b is inferred as F(4,4)
4  c = b + 1;
5  if(a + c == 0x12345678)
6  ...
```

```
1  ...
2  0x080484cd: mov  eax,DWORD PTR [ebp-0x14] ; load b
3  0x080484d0: add  eax,0x1                  ; b + 1
4  0x080484d3: mov  DWORD PTR [ebp-0x10],eax ; assign c
5  0x080484d6: mov  edx,DWORD PTR [ebp-0x18] ; load a
6  0x080484d9: mov  eax,DWORD PTR [ebp-0x10] ; load c
7  0x080484dc: add  eax,edx                  ; a + c
8  0x080484de: cmp  eax,0x12345678
9  ...
```

Listing 4: An example that shows the benefit of field-level symbolic emulation and removing `mov` instructions. (§3.4.1)

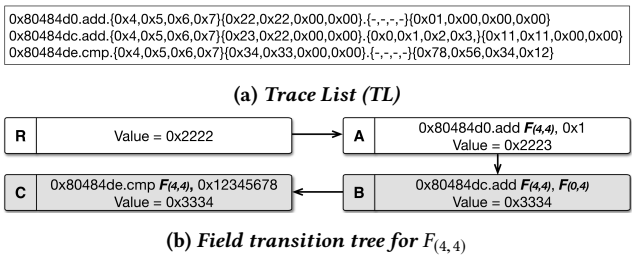


Figure 7: An example that inputs "a=0x1111" and "b=0x2222" to Listing 4. (a) TL. (b) Field transition tree. In TL, 0x1111 and 0x2222 represent "0x11,0x11" and "0x22,0x22", respectively. TL does not record `mov`-like instructions. The field transition tree was generated without `mov` instructions.

in the memory addresses and registers captured when the execution trace is generated. Figure 7 shows an example that infers a data flow from the execution trace. By running the code in Listing 4, the TL regarding $F_{(4,4)}$ is generated in all execution traces as shown in Figure 7a. As an operand of `cmp`, it is necessary to decide whether $F_{(4,4)}$ was directly used from an input or through an operation such as $F_{(4,4)}+1$ or $F_{(4,4)}+1+F_{(0,4)}$. Existing methods that manipulate symbolic tags, as shown in the assembly code of Listing 4, can precisely decide that $F_{(4,4)}$ was used by `cmp` through `ebp-0x14`→`eax`→ ... →`eax`; however, this incurs a large overhead. Instead, Intriguer makes this decision without incurring such an overhead: as an `add` instruction changes the values stored in memory and register, the TL records the concrete value changes. Intriguer can decide the instructions that $F_{(4,4)}$ (on `cmp`) has passed through, by tracing the changes in the input value obtained from the inferred field ($0x2222 \rightarrow 0x2223 \rightarrow 0x3334$).

In Figure 6-(c), Intriguer generates constraints by traversing the field transition tree: the uncomplicated constraints for depth =1 and the complicated constraints for depth >1 in the tree. In the field-level constraint solving step, Intriguer uses an SMT solver for the complicated constraints only, and directly solves the uncomplicated constraints. Note that the uncomplicated constraints that Intriguer can directly solve even include tight multi-byte constraints, such as magic bytes, thanks to our inferred-field approach. We describe more details of constraint solving in the following subsection.

Finally, to address the over-constraint problem in concolic execution, Intriguer uses a new method, similar to the optimistic solving of Qsym but more sophisticated, by manipulating the field transition tree. When adding a new node N_{new} to the tree, if the value V is the same as V of the root node as well as of another node N_{other}, then Intriguer appends this new node to the root node as well as to N_{other}. By doing this, Intriguer can generate simpler constraints in addition when generating constraints in the next step (§3.4), and thus it is possible to solve constraints that were not solved for over-constraint problems. Unlike the optimistic solving method of Qsym that solved the last constraint only, the field tree based method of Intriguer can more effectively deal with distinct states of the branch nodes, by adding more nodes to the new branch nodes.

3.4 Field-Level Constraint Solving

After constructing the field transition trees for all inferred fields, Intriguer performs field-level constraint solving. To do this, Intriguer conducts symbolic emulation on all types of instructions and uses a symbolic solver, similarly to the existing methods; but it traverses the field transition trees and picks out (un)complicated branch/arithmetic instructions, achieving great efficiency and effectiveness. Intriguer invokes a solver for complicated (truly complex) constraints only. Intriguer traverses all field transition trees and so obtains mutation values for all inferred fields. We explain more detail about symbolic emulation and constraint solving.

3.4.1 Symbolic Emulation. We explain three distinctive design strategies of Intriguer to optimize symbolic emulation. First, Intriguer symbolically emulates only the instructions that are relevant to the branches aimed to flip, by virtue of the field transition tree. Unlike our approach, existing concolic executors symbolically emulate all tainted instructions because they are unaware of the instructions needed for constraint solving. Note that Intriguer performs off-line concolic execution based on the execution trace, and picks out branch-relevant instructions from the "reduced" field-level trace list on the field transition tree (Figure 6) before the symbolic emulation. Second, Intriguer does not symbolically emulate `mov`-like instructions by removing them in the earlier phase. Note that it is possible to follow data flows without `mov`-like instructions by tracing changes in the value of the inferred fields (Figure 7a). Intriguer deals with symbolic expressions in this way, rendering more efficiency that reduces the emulation time of `mov`-like instructions (§2.1,§A). For example, Listing 4 shows in part an example C code and its assembly code receiving two four-byte integers. We could see that four `mov` instructions are used for executing two `add` instructions and one `cmp` instruction. Figure 7a and Figure 7b show the TL and the field transition tree generated by inputting `0x1111` and `0x2222` to *a* ($F_{(0,0)}$) and *b* ($F_{(4,4)}$), respectively. By emulating the nodes of the tree each by each, we could generate a symbolic expression comparing $(F_{(4,4)} + 0x1) + F_{(0,4)}$ and $0x12345678$ on a `cmp` instruction, i.e., without needing to emulate `mov` instructions. Third, Intriguer performs symbolic emulation at the field level. Unlike previous approaches manipulating symbolic expressions in the byte level, Intriguer operates symbolic expressions with regard to the inferred fields. For instance, existing methods generate and operate symbolic expressions for each byte of `eax`, i.e., four times,

but Intriguer manipulates a single symbolic expression by inferring these four bytes as a single field. Intriguer benefits from these new strategies with regard to performance in symbolic emulation.

3.4.2 Constraint Solving. Thanks to the inferred fields and their field transition trees, Intriguer can discover interesting offsets and values quickly to generate new test cases. Unlike other concolic execution engines, Intriguer generates constraints for arithmetic instructions (e.g., `add`, `mul`, `sub`) to trigger integer bugs as well as comparison instructions (e.g., `cmp`) to search for new paths. The intuition is that arithmetic boundary conditions can be dealt with like branch constraints. In addition, Intriguer classifies complicated and uncomplicated constraints according to the number of nodes constituting the constraints in the field transition tree. In doing so, unlike existing methods, Intriguer selectively uses an SMT solver (and also the symbolic emulation) for truly complex constraints only, and directly obtains the values that satisfy uncomplicated constraints, including both loose and tight multibyte constraints. Since uncomplicated constraints often dominate those constraints found in real programs (Figure 3), Intriguer can efficiently find interesting values by minimizing the use of an SMT solver (Table 17 in §B).

Figure 6-(d) shows the constraints being generated by traversing the field transition tree (c). Intriguer traverses all nodes of the field tree and generates constraints when the current node is a target instruction, i.e., a comparison instruction for branch boundary conditions or an arithmetic instruction for arithmetic boundary conditions. Constraints are generated by symbolically emulating instructions as in existing symbolic execution methods. Intriguer emulates all instructions in nodes from the current node that contains the target instruction to the root node. Note that the nodes in the path are the relevant nodes, meaning relevant instructions to the target instruction. At this time, constraints are generated according to the target instruction type (comparison or arithmetic). Thus, one or more constraints are generated from the four nodes using `cmp`, `add`, and `mul` instructions in this example.

Comparison instructions. Intriguer performs a constraint solving on comparison related instructions[5] to determine the branch conditions for finding a new path, just like previous concolic execution tools. Since there are two nodes with `cmp` instructions ("A: *addr1.* `cmp` *X*, 0x200" and "D: *addr7.* `cmp` *X*, 0x100") in Figure 6-(c), Intriguer generates the first and the fourth constraints (green) in Figure 6-(d) from this node. The first one is an uncomplicated constraint that will be directly solved for depth = 1: it is truly a simple condition that negates the comparison. The other one is a complicated constraint that will be solved by an SMT solver for depth = 3: the two conditions should be simultaneously met.

Arithmetic instructions. Arithmetic boundary values likely incur an overflow or underflow when running arithmetic instructions[6] as we discussed in §2.4. When encountering an arithmetic instruction like this, Intriguer adds, as a constraint, >INT_MAX (e.g., `0xffffffff` or `0x7fffffff`) for overflow checks and <0 for underflow checks, respectively, to the current node. Since there are two nodes, respectively, with `add` and `mul` instructions ("B: *addr4.* `add` *X*, 0x10" and "C: *addr6.* `mul` *X*, *Y*") in Figure 6-(c), Intriguer generates the second and the third constraints (red) in Figure 6-(d) from this node. These are the complicated constraint that will be solved by an SMT solver for depth = 2: both arithmetic and comparison conditions should be simultaneously met. Note that the latest advanced fuzzers (even Qsym) cannot deal with arithmetic boundary constraints effectively (Table 3) but Intriguer can do it.

Uncomplicated constraints. Intriguer decides uncomplicated constraints according to the depth in the field transition tree (depth = 1) and immediately resolves those uncomplicated constraints without querying the constraint solver. For example, Intriguer can deal with loose constraints (e.g., `x > 0`) and multibyte constraints (e.g., `x == 0x0f365c01`) as uncomplicated constraints, and exceptionally handle narrow range constraints (e.g., `x > 0` $\wedge$ `x < 10`). Thanks to the trace records provided by Intriguer's DTA Execution Monitor, it is possible to directly solve them. To do this, Intriguer regards a tainted operand as a variable, and an uncontrollable (untainted) operand as a constant. Intriguer then sets the variable as the constant value for a given comparison instruction, and performs three uncomplicated arithmetic operations with +0 for equality comparison cases (e.g., multibyte constraints and monotonous arithmetic constraints) and $\{-1, +1\}$ for inequality comparison cases (e.g., loose constraints). For the narrow range constraints as exceptionally above, Intriguer can obtain $\{-1, 1, 9, 11\}$ as boundary values. Note that inner values $\{1, 9\}$ accordingly satisfy the constraints, and Intriguer can resolve them without querying the constraint solver. Moreover, Intriguer directly deals with uncomplicated arithmetic constraints (e.g., $x + 1 >$ INT_MAX or $x - 1 < 0$) without using an SMT solver.

4 IMPLEMENTATION

We implement the execution monitor as a pintool written in 4.2K lines of C++ code for DTA using Pin 3.7, and also the core of Intriguer written in 2K lines of C++ code. We use Z3 (v.4.5) for constraint solving and boundary triggering in our prototype implementation. Finally we implement the fuzzer with AFL 2.41b. Intriguer can be easily ported to other OS environments since all of the tools (Pin 3.7, Z3, AFL) support the latest versions of Linux, Windows, and macOS. The current implementation supports most of 32-bit instructions and part of 64-bit instructions (e.g., pcmpeq). We will extend Intriguer to support more 64-bit instructions that are essential for bug discovery. Intriguer will be open-sourced and a mutation strategy that can be performed in field level will be added.

5 EVALUATION

To evaluate Intriguer, we set the following research questions.

- **RQ1:** How good is the performance of Intriguer in terms of bug detection capabilities? (§5.2)
- **RQ2:** By how much is code coverage increased by Intriguer? (§5.3)
- **RQ3:** Can Intriguer's *trace reduction* successfully remove execution traces without unintentionally removing interesting code blocks? (§5.4)
- **RQ4:** How effective is the *field transition tree* of Intriguer? (§5.5)
- **RQ5:** Can Intriguer's instrumentation of arithmetic boundary discover hard-to-trigger bugs which are missed by other fuzzers? (§5.6)

[5] We consider instructions such as `cmp`, `cmpsb`, `cmpsw`, `cmpsd`, `sub`, and `xor`.
[6] We consider instructions such as `add`, `mul`, `imul`, `shl`, and `sub`.

Table 6: Program data.

Program	# Lines	# Functions	# Branches	Seed Type
objdump	70,050	2,638	54,404	empty, elf, pe
nm	52,413	2,063	39,688	empty, elf
readelf	19,354	477	16,313	elf
ffmpeg	323,761	17,464	276,630	mp4
avconv	201,676	10,999	155,092	mp4
tiff2pdf	14,433	676	9,866	tiff
bsdtar	37,079	1,847	22,591	tar

Table 7: Seed data.

Type	Size (Byte)	Description
empty	1	A single NULL byte (0x00)
elf	7,868	Generated by GCC compiler (§D)
pe	46,080	Generated by Microsoft Visual C++ (§D)
mp4	5,614	Downloaded and minimized with ffmpeg
tiff	13,050(avg)	Downloaded and minimized with afl-cmin (69)
tar	2,048	Downloaded from FoRTE-Research

5.1 Evaluation Setup

To account for the random nature of fuzzing, we follow the recent guideline for evaluating fuzz testing [17]: we perform multiple trials and use statistical tests; evaluate different seeds; consider longer timeouts; and evaluate bug-finding performance using ground truth with benchmark test suites and real programs.

Programs. We used LAVA-M benchmark programs [11], and the real programs in GNU `binutils`, `ffmpeg`, `libav`, `libtiff`, and `libarchive`, for evaluation. The LAVA-M dataset is widely used for evaluation and comparison of recent fuzzers, including VUzzer and Qsym [8, 19, 27, 33]. Note that the binaries of LAVA-M involve various conditions including multi-byte constraints to get to buggy code; but to be free from concerns about overfitting problems, it is also very important to involve real-world programs, such as `ffmpeg`, in evaluation. GNU `binutils` [1] is a set of programs used to create or read binary programs, object files, and libraries. `ffmpeg` and `libav` [2] are free software tools and library that processes audio and media files. `libtiff` [18] is a free software library that processes tiff images. `libarchive` is a library that reads and writes streaming archives. Recent studies and even commercial fuzzers have used these real-world programs for bug discoveries [4, 12, 27, 33]. Table 6 summarizes the program data.

Fuzzers. We set baseline fuzzers for evaluation: Qsym and VUzzer for benchmarks in LAVA-M (§5.2.1); and Qsym and AFL for bug discoveries in real-world programs (§5.2.2) and for code coverage (§5.3). In LAVA-M experiments, Intriguer and VUzzer use a single core, and Qsym uses three cores for hybrid fuzzing. In real program experiments, Intriguer and Qsym both perform hybrid fuzzing.

Seeds. The executable format (e.g., elf or pe) files are binaries compiled from 22 lines of C++ code containing a simple class, using g++ and visual studio (§D). The mp4 file was downloaded from the sample-videos[7] site and left with only two video frames, reducing the repeated structures by `ffmpeg`. The tiff files were downloaded from go-fuzz-corpus[8] repository. We reduced 215 downloaded files to 69 files by `afl-cmin`. The tar file was downloaded from FoRTE-Research [24]. Table 7 summarizes the seed data.

[7] https://sample-videos.com/

Table 8: Number of bugs detected by the latest fuzzers and Intriguer. We report the median of 20 runs per fuzzer and also the maximum. Note that Intriguer detected all the bugs (2,136 bugs) in `who`. (5h) Readers are referred to Table 16 in §E for more comparisons.

Program	#Bugs (listed)	median			max		
		VUzz.	Qsym	**Intr.**	VUzz.	Qsym	**Intr.**
uniq	28	28	28	**28**	28	28	**28**
base64	44	39	44	**44**	40	44	**44**
md5sum	57	23	57	**57**	26	57	**57**
who	2,136	10	1,396	**2,131**	29	1,452	**2,136**
Total	2,265	100	1,525	**2,260**	123	1,581	**2,265**

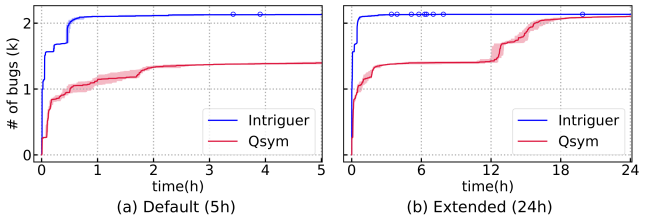


Figure 8: The cumulative number of bugs detected in LAVA-M's `who` over time. (a) Default 5-hour run; (b) Extended 24-hour run. The solid lines are median. The shaded region around is the confidence interval. The empty circle is the time that all the bug was detected. We use the Mann Whitney U-test, which is non-parametric, between Intriguer and Qsym for 20 runs ($p < 10^{-7}$).

Platform and configuration. We conducted our experiments on a machine with 2.40GHz Xeon CPU and 384GB RAM, running 64bit Ubuntu 16.04 systems. In the experiments, we conformed to the following comparison configuration: the LAVA-M dataset was fuzzed for default five hours (and particularly `who` in 5h and 24h) (§5.2.1), while real-world programs were fuzzed for 24 hours (§5.2.2, §5.3, §5.4, §5.5, and §5.6). Additionally, we performed 5-day fuzzing for `ffmpeg` (§5.2.2).

5.2 RQ1: Bug Discovery Capability

To answer RQ1, we compare the followings with other fuzzers: (i) the number of bugs identified in LAVA-M, and (ii) the number of distinct bugs discovered in real-world programs. As for distinct bugs, we performed triage based on a unique patch when the programs were patched [17]. Otherwise, we manually ran deduplication again after the triage based on a call stack.

5.2.1 LAVA-M Dataset. To compare the performance of Intriguer with VUzzer (i.e. taint-based fuzzer) and Qsym (i.e. concolic-based hybrid fuzzer), we performed 20 runs of fuzzing for each fuzzer on the buggy programs of LAVA-M. Table 8 shows the results of 5-hour fuzzing on each program of the LAVA-M dataset. The first two columns respectively show the target program names and the

[8] https://github.com/dvyukov/go-fuzz-corpus

Table 9: The number of bugs found by Intriguer, Qsym, and AFL, respectively, in real-world programs. (24h) We state the total number of 20 runs. Parentheses mean the number of bugs that the other two fuzzers could not find.

Program	Seed	AFL	Qsym	Intriguer
objdump	empty	1 (0)	3 (0)	9 (6)
	elf	5 (0)	7 (1)	11 (5)
	pe	4 (0)	7 (3)	9 (5)
nm	empty	3 (0)	3 (0)	4 (1)
	elf	3 (0)	3 (0)	4 (1)
readelf	elf	4 (0)	4 (0)	6 (2)
ffmpeg	mp4	0 (0)	1 (0)	2 (1)
avconv	mp4	0 (0)	5 (3)	4 (2)
tiff2pdf	tiff	0 (0)	0 (0)	1 (1)
bsdtar	tar	0 (0)	0 (0)	0 (0)
total	-	20 (0)	33 (7)	50 (24)

number of artificial bugs listed[9] by the LAVA authors [11]. The middle columns show the median values in the number of bugs identified by each fuzzer, and the last three columns the maximum.

The LAVA-M benchmark results show that Intriguer outperforms the state-of-the-art fuzzers. Qsym and VUzzer have difficulty finding all the bugs hidden behind multibyte constraints within the time budget. VUzzer detected only 4.4% of bugs and Qsym detected 58.2% of bugs, respectively, once in 20 runs of our experiments. On the contrary, Intriguer found 100% of bugs within the same time budget during our tests, two times in 20 runs; and when running under the same conditions for 24 hours, Intriguer found all the bugs nine times in 20 runs (Figure 8). Intriguer also found unlisted bugs although we don't show them in Table 8, e.g., in `who`, Intriguer identified 2464 bugs when found all listed bugs (2136 + 328).

Figure 8 depicts the number of listed bugs found by Intriguer and Qsym, respectively, in `who` over time. Intriguer found more than 1,500 bugs in 5m; but Qsym slowed down in finding bugs after 2h and found fewer than 1,500 bugs in 5h. Qsym was able to find more bugs after 12h, but never succeeded in finding all the bugs within 24h. This result indicates that the field-level constraint solving approach of Intriguer is effective in discovering new test cases and bugs in a shorter time.

5.2.2 Real-world Programs. In all programs, Intriguer found more bugs than AFL. This indicates that Intriguer solves the complicated constraints that the coverage-based fuzzer (AFL) could not solve, and creates new test cases, consequently finding more bugs. In addition, compared to Qsym, Intriguer found more bugs in `objdump`, `nm`, `readelf`, `ffmpeg`, and `tiff2pdf`. This indicates that Intriguer can discover bugs more efficiently by performing trace reduction and field-level constraint solving to optimize symbolic emulation. Only in `avconv`, Qsym found more bugs than Intriguer while they respectively found three and two unique bugs. We observe that this result was due to the Qsym's slightly different seed prioritization strategy, in which the most recently generated test case is selected among test cases having the same priority.

[9] The LAVA-M programs contain additional "unlisted" bugs, also identified by Intriguer.

Table 10: New bugs discovered in real-world programs.

Project	Bug Type	Report ID	Fixed
binutils	Out-of-bounds Read	22307	✓
binutils	Integer Overflow	22373	✓
binutils	Integer Overflow	22376	✓
binutils	Out-of-bounds Read	22384	✓
binutils	Integer Overflow	22385	✓
binutils	Integer Overflow	22386	✓
binutils	Out-of-bounds Read	22443	✓
binutils	Out-of-bounds Read	22506	✓
binutils	Integer Overflow	22507	✓
binutils	Out-of-bounds Write	22508	✓
binutils	Null Pointer Dereference	22509	✓
binutils	Null Pointer Dereference	22510	✓
binutils	Integer Overflow	22809	✓
binutils	Out-of-bounds Read	23147	✓
binutils	Out-of-bounds Read	23148	✓
binutils	Negative-size-param	23316	✓
binutils	Out-of-bounds Read	24266	✓
binutils	Out-of-bounds Read	24272	✓
binutils	Out-of-bounds Read	24273	✓
binutils	Out-of-bounds Read	24898	✓
binutils	Floating Point Exception	24921	✓
binutils	Out-of-bounds Read	24922	✓
ffmpeg	Out-of-bounds Write	mailing list	✓
libav	Integer Overflow	1088	
libav	Null Pointer Dereference	1089	✓
libav	Out-of-bounds Read	1093	
libav	Out-of-bounds Read	1094	
libav	Out-of-bounds Read	1095	
libav	Null Pointer Dereference	1099	
libav	Out-of-bounds Write	1100	✓
libav	Null Pointer Dereference	1101	✓
libav	Out-of-bounds Read	1104	
libav	Out-of-bounds Read	1105	
libav	Out-of-bounds Read	1106	
libav	Out-of-bounds Read	1143	
libav	Out-of-bounds Write	1144	
libav	Out-of-bounds Read	1145	
libav	Out-of-bounds Write	1146	
libav	Out-of-bounds Read	1147	
libav	Out-of-bounds Read	1148	
libav	Out-of-bounds Read	1149	
libav	Floating Point Exception	1150	
libtiff	Out-of-bounds Read	2849	

We performed fuzz testing for five days on `ffmpeg`, which contains more branches than the other programs. Intriguer discovered a new bug in `ffmpeg` that was not found by Qsym. The bug that Intriguer found in `ffmpeg` was latent in `ffmpeg` v2.5, which was released in December 15, 2014, for four years. Note that `ffmpeg` is a program that the state-of-the-art fuzzers like OSS-Fuzz [28] have intensively tested. This result show that previous fuzzers could not

Table 11: The number of branches found by fuzzers in 24h fuzzing. We state the median values of 20 runs along with p values with statistical tests of Intriguer vs. AFL and Qsym.

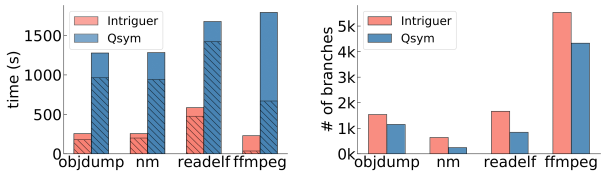
Program	Seed	Fuzzer	Branches	Factor	p
objdump	empty	Intriguer	2950.5	-	-
		Qsym	3207.0	1.087	< 0.002
		AFL	2499.5	0.847	< 0.001
	elf	Intriguer	3296.0	-	-
		Qsym	2634.0	0.799	$< 10^{-6}$
		AFL	2240.0	0.68	$< 10^{-7}$
	pe	Intriguer	3050.5	-	-
		Qsym	3142.5	1.03	< 0.049
		AFL	2350.5	0.771	$< 10^{-7}$
nm	empty	Intriguer	2848.5	-	-
		Qsym	2175.0	0.764	0.131
		AFL	1386.5	0.487	$< 10^{-5}$
	elf	Intriguer	2826.5	-	-
		Qsym	2637.0	0.933	$< 10^{-5}$
		AFL	2324.5	0.822	$< 10^{-7}$
readelf	elf	Intriguer	6823.0	-	-
		Qsym	6011.5	0.881	$< 10^{-7}$
		AFL	5437.0	0.797	$< 10^{-7}$
ffmpeg	mp4	Intriguer	31420.0	-	-
		Qsym	19242.5	0.612	$< 10^{-7}$
		AFL	13452.0	0.428	$< 10^{-7}$
avconv	mp4	Intriguer	18222.0	-	-
		Qsym	15391.0	0.845	$< 10^{-3}$
		AFL	12614.0	0.692	$< 10^{-7}$
tiff2pdf	tiff	Intriguer	2351.0	-	-
		Qsym	2324.0	0.989	< 0.002
		AFL	2284.5	0.972	$< 10^{-5}$
bsdtar	tar	Intriguer	2623.0	-	-
		Qsym	2424.5	0.924	< 0.002
		AFL	1856.5	0.708	$< 10^{-7}$

pass through highly constrained branches while existing concolic execution based fuzzers could not run the corresponding code because of longer execution time. On the contrary, Intriguer was able to find new execution paths efficiently through trace reduction and field-level constraint solving, while preserving the effectiveness of concolic based approach. Table 10 lists the bugs newly discovered and reported by Intriguer. Intriguer found 22 and 19 new bugs in `binutils` and `libav`, respectively, and also found new bugs in `ffmpeg` and `libtiff`.

Based on the observation of Figure 8, and Tables 8, 9, and 10, we can see that Intriguer detects many bugs both in LAVA and real-world programs. We can positively answer **RQ1**.

5.3 RQ2: Code Coverage

To answer RQ2, we examined the ability of fuzzers (Intriguer, Qsym, and AFL) to discover new branches in 24h fuzzing. We used *gcov* to measure branch coverage. Table 11 shows the number of newly discovered branches as a result of 24h fuzzing on seven real-world



(a) *time* **(b) *coverage***

Figure 9: New branches discovered by executing 20 AFL-generated test cases in Intriguer and Qsym, respectively. (a) Time taken for execution. (b) Number of new branches. The shaded part of each bar shows the solving time and the solid part is the emulation time. Intriguer is faster than Qsym, and also shows the higher coverage.

Table 12: The execution trace size and the execution time reduced by Intriguer's trace reduction using different threshold values.

Program	Trace Size (k lines)				Time (s)			
objdump	26.9	36.9	37.1	52.8	42	59	59	75
nm	4.6	8.8	11.1	19.0	30	48	49	62
readelf	21.9	38.8	46.3	66.9	63	147	148	167
ffmpeg	728.4	729.8	734.3	875.5	87	89	104	165
avconv	261.7	264.9	269.5	425.3	34	35	41	188
tiff2pdf	13.2	14.3	16.4	30.6	40	41	41	48
threshold	16	32	64	none	16	32	64	none

programs. If the factor is greater than 1, it means the corresponding fuzzer finds more branches than Intriguer. If the factor is less than 1, it means the corresponding fuzzer finds fewer branches than Intriguer.

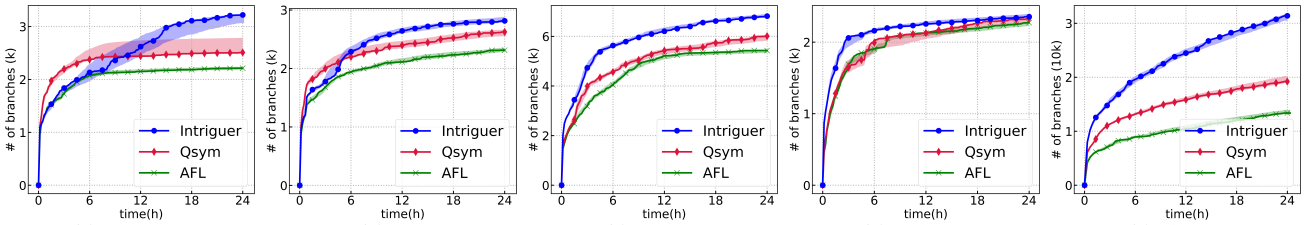
In every case, Intriguer and Qsym achieved the higher branch coverage than AFL. The constraint solver solves constraints that fuzzers were not able to solve in hybrid fuzzing, and helps them discover newer branches. Except for `objdump` (empty and pe format), Intriguer achieved greater branch coverage than Qsym. Figure 10 shows branch coverage in `objdump`, `nm`, `readelf`, `tiff2pdf` and `ffmpeg` by 24h experiments. We can see that Intriguer outperformed both Qsym and AFL on most cases.

In Figure 9, we used 20 test cases generated by AFL in evaluating execution time and branch coverage of Intriguer and Qsym, respectively. We set the timeout as 90s for both Intriguer and Qsym. The measured execution time consists of solving time and emulation time. As depicted in Figure 9, Intriguer outperforms Qsym with regard to both solving time and emulation time, enabling faster execution and higher coverage. In particular, large programs, such as `ffmpeg`, have significantly reduced execution time and achieved greater coverage.

Based on the observation of Table 11, and Figures 9 and 10, we can see that Intriguer increases the code coverage of the target program. We can positively answer **RQ2**.

5.4 RQ3: Execution Trace Reduction

To evaluate the effectiveness of trace reduction, we compared the size of execution traces and the execution time using different threshold values in trace reduction, on the programs shown in



(a) ***objdump*** $p_1 < 10^{-6}$, $p_2 < 10^{-7}$ (b) ***nm*** $p_1 < 10^{-5}$, $p_2 < 10^{-7}$ (c) ***readelf*** $p_1 < 10^{-7}$, $p_2 < 10^{-7}$ (d) ***tiff2pdf*** $p_1 < 0.002$, $p_2 < 10^{-5}$ (e) ***ffmpeg*** $p_1 < 10^{-7}$, $p_2 < 10^{-7}$

Figure 10: The number of newly discovered branches in each program execution by 24h fuzzing of Intriguer, Qsym, and AFL, respectively. p_1 is a p-value of Intriguer and Qsym, and p_2 is of Intriguer and AFL.

Table 13: The number of newly found branches and bugs when Intriguer performs fuzzing with/without trace reduction in 24h.

Program	# of branches (median)			# of bugs (total)	
	w/ R	w/o R	p	w/ R	w/o R
objdump	3296.0	3068.5	$< 10^{-4}$	11 (6)	5 (0)
nm	2826.5	2797.5	$= 0.039$	4 (1)	3 (0)
readelf	6823.0	6381.5	$< 10^{-6}$	6 (2)	4 (0)
ffmpeg	31420.0	22699.0	$< 10^{-7}$	2 (1)	1 (0)
avconv	18222.0	11372.5	$< 10^{-7}$	4 (3)	1 (0)
tiff2pdf	2351.0	2354.5	$= 0.186$	1 (0)	1 (0)

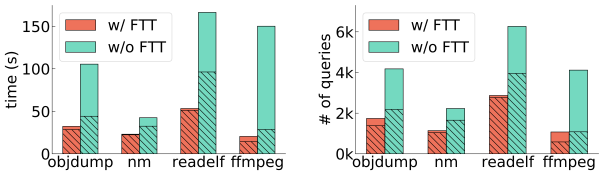
Table 12. We used various threshold values (16, 32, 64) and also skipped (none) the trace reduction step for comparisons. We observe that with a threshold of 16, the trace size and execution time are maximally reduced: the execution time decreased by at most 82% (`avconv`) and at least 45% (`ffmpeg`). Interestingly, `ffmpeg` showed a relatively small percentage of reduction in execution traces (20% in `ffmpeg` and 51% in the third lowest `objdump`), but more rapid reduction in execution time (similar rates to `objdump`). We observe that `ffmpeg` more frequently used encoding/decoding operations accessing a wide range of input offsets than `objdump`, and related complex constraints were successfully removed.

To find out if trace reduction is effective and bugs are not missed by the reduction of important traces, we performed 24h fuzzing with and without trace reduction. In Table 13, we observe that fuzzing with trace reduction found significantly more branches and more bugs than fuzzing without trace reduction, except for `tiff2pdf`. Regarding false negatives, our experiments showed that there was no bug missed by using trace reduction in all programs; and instead newer test cases and more bugs were found within the same amount of time.

Based on the observation from Table 12 and 13, we see that Intriguer successfully reduces execution time with a small false negative. These observations allow us to positively answer **RQ3**.

5.5 RQ4: Field Transition Tree

Intriguer performs field-level constraint solving by constructing a field transition tree from the FL for each inferred field. To evaluate the effectiveness of constructing the field transition trees, we performed fuzz testing by dividing two cases of constraint generation,



(a) ***time*** (b) ***queries***

Figure 11: Field-level constraint solving done in a single execution of Intriguer by directly feeding a seed file, with/without field transition trees. (a) Time taken for field-level constraint solving. (b) Number of queries used. In each bar, the shaded part is arithmetic constraints, and the solid part is branch constraints, showing that the branch constraints are more reduced with field transition trees.

with and without constructing the trees. Note that in the latter case, the node's *Value* checking step of the tree construction phase is unavailable, making all trace records in the FL (overly) used for constraint solving. In addition, unlike the field transition trees that already involve branches, the FL can only provide a form of lists. Thus, the constraint is not generated for each branch according to the field value. Instead, the constraint is generated by using all the nodes executed before the current node, causing over-constraints possibly to occur. This is one of the strong reasons why we decided to construct field transition trees for each inferred field to avoid the latter case.

Table 14 shows the number of branches and bugs discovered by Intriguer with or without field transition trees in 24h fuzzing (20 runs). The results indicate that the case with field transition trees discovers significantly more branches in all programs and more bugs (in `objdump`, `nm`, `readelf`, and `avconv`) than the case without trees for the same amount of time.

Figure 11 shows the time taken for constraint solving by a symbolic solver in a single execution of Intriguer (without fuzzing, i.e., only the Execution Monitor and the Core as shaded in Figure 4) by directly feeding a seed file, and the number of resulting queries. Note that the field transition trees enable us to tune constraints to uncomplicated (depth = 1) and complicated (depth > 1) constraints, and the complicated constraints only to be solved by a symbolic solver. The results indicate that constructing the field transition trees helps to perform constraint solving in a shorter time and with reduced queries; and consequently discover more branches and bugs. Branch constraints by comparison instructions were significantly more reduced than arithmetic constraints when the field

Table 14: The number of newly found branches and bugs when Intriguer performs fuzzing with/without field transition trees in 24h.

Program	# of branches (median)			# of bugs (total)	
	w/ FTT	w/o FTT	p	w/ FTT	w/o FTT
objdump	3296.0	3157.5	$< 10^{-3}$	11 (5)	7 (1)
nm	2826.5	2745.5	$= 0.07$	4 (1)	3 (0)
readelf	6823.0	6297.5	$< 10^{-7}$	6 (2)	4 (0)
ffmpeg	31432.0	27108.0	< 0.002	2 (0)	2 (0)
avconv	18222.0	16217.0	$< 10^{-3}$	4 (1)	3 (0)
tiff2pdf	2351.0	2308.5	< 0.003	1 (0)	1 (0)

transition tree was used. Note that more branches and bugs found (Table 14) and less time and queries needed (Figure 11) both imply that the field transition trees successfully suppress mistaints and over-constraints. In addition, the field transition trees are effectively used to tune those constraints.

Based on the observation from Table 14 and Figure 11, we see that the field transition tree significantly increases new branches and bugs discovered in most cases. These observations allow us to positively answer **RQ4**.

5.6 RQ5: Arithmetic Boundary Instrumentation

Table 15 shows the number of new branches and bugs discovered by Intriguer with or without arithmetic boundary instrumentation in 24h fuzzing (20 runs). When using arithmetic boundary instrumentation in `objdump`, `readelf`, `avconv` and `tiff2pdf`, Intriguer discovered bugs that were not found when not using it. Among these, three bugs in `objdump`, `readelf`, and `tiff2pdf`, and two bugs in `avconv` were not found by both Qsym and AFL. In `objdump`, we observe that two bugs were found only when arithmetic boundary instrumentation was not used: this result is due to more test cases generated within the same time budget. If we enhance fuzzing time, then it is possible to find the missed bugs when using arithmetic boundary instrumentation; but the reverse case is unlikely because bugs found only when using arithmetic boundary instrumentation are hard to trigger — triggered only when a specific bug trigger condition is met. Moreover, when using arithmetic boundary instrumentation, Intriguer was able to find more branches than not using it.

Based on the observation from Table 15, we see that the arithmetic boundary instrumentation significantly increases new branches discovered in most cases and finds hard-to-trigger bugs. This observation allow us to positively answer **RQ5**.

6 DISCUSSION

In this paper, we aimed to present a new mutation-based hybrid fuzzer which is efficient and effective in triggering both hard-to-reach bugs and hard-to-trigger bugs, with optimization based on inferred field knowledge in symbolic execution. Our field-level constraint solving concept is well-tuned by dynamic taint analysis and

Table 15: The number of newly found branches and bugs when Intriguer performs fuzzing with/without arithmetic boundary instrumentation in 24h.

Program	# of branches (median)			# of bugs (total)	
	w/ ABI	w/o ABI	p	w/ ABI	w/o ABI
objdump	3296.0	3140.5	$< 10^{-3}$	11 (1)	12 (2)
nm	2826.5	2695.5	$< 10^{-3}$	4 (0)	4 (0)
readelf	6823.0	6571.0	$< 10^{-5}$	6 (1)	5 (0)
ffmpeg	31420.0	25513.0	$< 10^{-6}$	2 (0)	2 (0)
avconv	18222.0	13062.5	$< 10^{-7}$	4 (3)	1 (0)
tiff2pdf	2351.0	2374.0	$= 0.15$	1 (1)	0 (0)

the minimal use of a constraint solver for truly complex constraints. We discuss several interesting issues regarding our approach.

Intriguer introduced field-level constraint solving that efficiently derives interesting mutation values from the reduced execution traces of each inferred field; however, Intriguer performed field inference by inspecting offsets recorded in the execution traces and did not consider the association between those fields. For example, if the same data structure is repeated in the input, the inferred fields, F_a and F_b, could be field A of the same type in the execution traces. Since Intriguer constructs a field transition tree for all inferred fields if the same field A is used in multiple locations, the same field A will be inferred at multiple offsets, therefore it will construct multiple trees and the result in repetition. To perform field-level constraint solving more efficiently, Intriguer can go one step further in offset-based field inference, identifying fields of the same type through the FL and grouping them together. In addition to identifying repeated fields, we may infer structure from those fields and then consider mutation strategies specific to repeated fields based on it.

Intriguer reduces the execution traces to emulate only the small portion of the instructions that are repeatedly used to access a wide range of input bytes. One may have two concerns regarding trace reduction. First, excessively reduced traces may affect a constraint solving for important branches. Although we already examined the performance and addressed the false negative concerns in §5.4, we could consider further steps toward this concern: it is possible to gradually increase the threshold value of trace reduction if the fuzzer can no longer finds new test cases when performing hybrid fuzzing. Moreover, we could consider the context-sensitivity of the run-time process, which was also considered in Angora [8] and Qsym [33]. Context-sensitivity means that Intriguer will not perform trace reduction when a program is in a different context (e.g., different call stack). Second, the instructions can be repeatedly used to access only a narrow range of input bytes. Note that an execution bottleneck can also occur if the instructions are used to repetitively access the input with specific offsets. We can address this problem by reducing the execution traces for the instructions that use the same offset by considering the program's context.

Intriguer currently supports most of the x86 instruction set and a part of x86_64 instruction set. Although it is challenging to support all of x86_64 instructions, we can implement frequently used instructions to support actual program execution. Note that Intriguer uses only reproducible traces in symbolic execution through field transition trees, and thus it is possible to minimize the problem

caused by unimplemented instructions, e.g., trying to solve incomplete constraints. Furthermore, Intriguer requires less implementation effort than other instruction-level concolic execution engines because it does not require symbolic execution handlers for data transfer instructions.

Since Intriguer uses DTA, taint-related problems should be discussed. Due to the large number of instructions, it is very difficult in practice to implement handlers for all instructions and to process highly complex instructions precisely. Accordingly, DTA is susceptible to mistainting problems: the mistainted values and their instructions could incur unnecessary operations and wasteful constraint queries in the constraint solving step. Therefore, Intriguer addresses this problem by constructing the field transition tree as we described in §3.3.3. Moreover, the execution monitor of Intriguer is already developed to deal with known over-tainting problems (§C). However, there is an over-tainting problem regarding an operand value due to over-tainting in the execution trace: if the value is incorrect, then the traces are removed by the field transition tree again; but if it is correct, the traces will remain. We leave this limitation as a future study.

Finally, in DTA, there is a case that data is indirectly propagated (e.g., indirect and conditional jump), compared to the usual case that data is directly propagated (e.g., data transfer instruction). Since it is unlikely to reproduce the indirectly propagated data from input values, the traces of indirect propagation may not be added to field transition trees. The current version of Intriguer does not consider the control-flow dependency, e.g., occurring from indirect and conditional jump, and we also leave this limitation as a future study.

7 RELATED WORK

7.1 Coverage-based Fuzzing

Fuzzing can be guided by sophisticated program analysis techniques [7, 13, 15, 30]. Coverage-based fuzzers like AFL [34] attempt to increase code coverage by prioritizing inputs that likely explore new paths through the guidance of coverage information collected during program execution. AFLFast [4] leverages the Markov model based technique to prioritize low-frequency paths, and AFLGo [3] minimizes the distance of seeds to targets. CollAFL [12] attempts to provide more accurate coverage information. AFLFast and AFLGo prioritize test cases and perform fuzzing longer for test cases of higher priority.

However, they lack the penetration power: fuzzers get stuck when encountering multibyte constraints. To resolve this problem, program analysis techniques have been frequently used. VUzzer [27] adopted application-aware fuzzing techniques that leverage both static and dynamic program analysis. VUzzer is credited for immediate treatment of multibyte constraints through taint tracking and static analysis; but it incurs false positives in magic byte detection and requires many valid initial inputs. Steelix [19] leverages light-weight static analysis and binary instrumentation to provide not only coverage information but also comparison progress information to a fuzzer, for multibyte comparisons. Steelix is more efficient than approaches based on taint analysis but less effective at detecting a new execution path when complex constraints are faced. Angora [8] combines byte-level taint tracking, shape and type inference, and search techniques to solve path constraints fast. These approaches are, however, limited to dealing with certain types of constraints only, as explained in [33]. Unlike these approaches, Intriguer queries a constraint solver, but for truly complex constraints only. T-Fuzz [26] removes complex constraint checks by program transformation and runs symbolic execution to filter out false positives. It negates interesting checks in the target program by binary rewriting. To reproduce true bugs in the original program, however, T-Fuzz has to filter out false positives in the transformed program and so uses a symbolic analysis, incurring program explosion problems. We compared Intriguer with VUzzer by own experiment and with Steelix, Angora, and T-Fuzz by reports, in the LAVA-M benchmark dataset.

7.2 Hybrid Fuzzing

Combining both fuzzing and concolic execution is a promising direction to addressing the problem of penetration power in fuzzing. This hybrid fuzzing concept was introduced [6, 20, 25], and adopted by the latest fuzzers. Driller [29] uses fuzzing and selective concolic execution in a complementary manner to explore deeper paths by selectively invoking concolic execution when fuzzing gets stuck. However, previous hybrid fuzzers including Driller use general concolic executors, and so they are slow and also susceptible to path explosions. Qsym [33] raised the performance bottleneck problem in symbolic emulation, and significantly improved the performance of hybrid fuzzing by developing a scalable concolic execution engine with instruction-level emulation and various heuristics such as optimistic solving and basic block pruning. Qsym is remarkable in terms of its design and implementation.

8 CONCLUSIONS

This paper presented Intriguer, which is efficient and effective in finding both hard-to-reach bugs and hard-to-trigger bugs in programs. Intriguer's key idea, which came from our systematic analysis of hybrid fuzzing, is field-level constraint solving. Unlike existing hybrid fuzzers, such as Driller and Qsym, Intriguer minimally uses a solver for truly complex constraints only and significantly reduces symbolic emulation overhead. Unlike program analysis based techniques, such as VUzzer and Steelix, Intriguer leverages a constraint solver, allowing more chances to solve complex constraints. The practical performance of field-level constraint solving renders great opportunities to hybrid fuzzing. Our evaluation results showed that Intriguer outperformed Qsym and VUzzer in the LAVA-M testset; and more importantly, found 43 new bugs in real-world programs, including the new bug in `ffmpeg` that has been missed by OSS-Fuzz for four years, and received 23 new CVEs.

ACKNOWLEDGMENTS

We thank the anonymous reviewers and our shepherd Andrew Ruef for helpful comments and suggestions on this work. This research was supported in part by the Institute for Information & communications Technology Promotion (IITP) grant funded by the Korea government (MSIT) (No.2018-0-00513, Machine Learning Based Automation of Vulnerability Detection on Unix-based Kernel).

REFERENCES

[1] 2018. GNU Binutils. https://www.gnu.org/software/binutils/index.html
[2] 2018. Libav Open source audio and video processing tools. https://libav.org/
[3] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed Greybox Fuzzing. In *Proc. the ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM.
[4] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2016. Coverage-based Greybox Fuzzing as Markov Chain. In *Proc. the ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 1032–1043.
[5] Juan Caballero, Heng Yin, Zhenkai Liang, and Dawn Song. 2007. Polyglot: Automatic Extraction of Protocol Message Format Using Dynamic Binary Analysis. In *Proc. the ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 317–329.
[6] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. 2012. Unleashing MAYHEM on Binary Code. In *Proc. the IEEE Symposium on Security and Privacy (S&P)*. IEEE, 380–394.
[7] Hongxu Chen, Yinxing Xue, Yuekang Li, Bihuan Chen, Xiaofei Xie, Xiuheng Wu, and Yang Liu. 2018. Hawkeye: Towards a Desired Directed Grey-box Fuzzer. In *Proc. the ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 2095–2108.
[8] Peng Chen and Chen Hao. 2018. Angora: Efficient Fuzzing by Principled Search. In *Proc. the IEEE Symposium on Security and Privacy (S&P)*. IEEE.
[9] Paolo Milani Comparetti, Gilbert Wondracek, Christopher Kruegel, and Engin Kirda. 2009. Prospex: Protocol Specification Extraction. In *Proc. the IEEE Symposium on Security and Privacy (S&P)*. IEEE, 110–125.
[10] Weidong Cui, Marcus Peinado, Karl Chen, Helen J Wang, and Luis Irun-Briz. 2008. Tupni: Automatic Reverse Engineering of Input Formats. In *Proc. the ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 391–402.
[11] Brendan Dolan-Gavitt, Patrick Hulin, Engin Kirda, Tim Leek, Andrea Mambretti, Wil Robertson, Frederick Ulrich, and Ryan Whelan. 2016. Lava: Large-scale Automated Vulnerability Addition. In *Proc. the IEEE Symposium on Security and Privacy (S&P)*. IEEE, 110–121.
[12] Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuwen Tu, Kang Li, Zhongyu Pei, and Zuoning Chen. 2018. CollAFL: Path Sensitive Fuzzing. In *Proc. the IEEE Symposium on Security and Privacy (S&P)*. IEEE, 679–696.
[13] Vijay Ganesh, Tim Leek, and Martin Rinard. 2009. Taint-based Directed Whitebox Fuzzing. In *Proc. the International Conference on Software Engineering (ICSE)*. 474–484.
[14] Patrice Godefroid, Michael Y Levin, David A Molnar, et al. 2008. Automated Whitebox Fuzz Testing. In *Proc. the Network and Distributed System Security Symposium (NDSS)*, Vol. 8. 151–166.
[15] Istvan Haller, Asia Slowinska, Matthias Neugschwandtner, and Herbert Bos. 2013. Dowsing for Overflows: A Guided Fuzzer to Find Buffer Boundary Violations. In *Proc. the USENIX Security Symposium (SEC)*. USENIX Association, 49–64.
[16] Vasileios P Kemerlis, Georgios Portokalidis, Kangkook Jee, and Angelos D Keromytis. 2012. libdft: Practical Dynamic Data Flow Tracking for Commodity Systems. In *Acm Sigplan Notices*, Vol. 47. ACM, 121–132.
[17] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating Fuzz Testing. In *Proc. the ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 2123–2138.
[18] Sam Leffler. 1999. LibTIFF–TIFF Library and Utilities. http://www.libtiff.org/
[19] Yuekang Li, Bihuan Chen, Mahinthan Chandramohan, Shang-Wei Lin, Yang Liu, and Alwen Tiu. 2017. Steelix: Program-state Based Binary Fuzzing. In *Proc. the Joint Meeting on Foundations of Software Engineering (FSE)*. ACM, 627–637.
[20] Rupak Majumdar and Koushik Sen. 2007. Hybrid Concolic Testing. In *Proc. the International Conference on Software Engineering (ICSE)*. IEEE, 416–426.
[21] Barton P Miller, Louis Fredriksen, and Bryan So. 1990. An Empirical Study of the Reliability of UNIX Utilities. *Commun. ACM* 33, 12 (1990), 32–44.
[22] Jiang Ming, Dinghao Wu, Gaoyao Xiao, Jun Wang, and Peng Liu. 2015. TaintPipe: Pipelined Symbolic Taint Analysis. In *Proc. the USENIX Security Symposium (SEC)*. USENIX Association, 65–80.
[23] David Molnar, Xue Cong Li, and David Wagner. 2009. Dynamic Test Generation to Find Integer Bugs in x86 Binary Linux Programs. In *Proc. the USENIX Security Symposium*. USENIX Association.
[24] Stefan Nagy and Matthew Hicks. 2019. Full-speed Fuzzing: Reducing Fuzzing Overhead through Coverage-guided Tracing. In *Proc. the IEEE Symposium on Security and Privacy (S&P)*. IEEE.
[25] Brian S Pak. 2012. Hybrid Fuzz testing: Discovering Software Bugs via Fuzzing and Symbolic Execution. *School of Computer Science Carnegie Mellon University* (2012).
[26] Hui Peng, Yan Shoshitaishvili, and Mathias Payer. 2018. T-Fuzz: fuzzing by program transformation. In *Proc. the IEEE Symposium on Security and Privacy (S&P)*. IEEE, 697–710.
[27] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. 2017. VUzzer: Application-aware Evolutionary Fuzzing. In *Proc. the Network and Distributed System Security Symposium (NDSS)*.
[28] Kostya Serebryany. 2017. OSS-Fuzz-Google's continuous fuzzing service for open source software. (2017).
[29] 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 *Proc. the Network and Distributed System Security Symposium (NDSS)*.
[30] Tielei Wang, Tao Wei, Guofei Gu, and Wei Zou. 2010. TaintScope: A Checksum-Aware Directed Fuzzing Tool for Automatic Software Vulnerability Detection. In *Proc. the IEEE Symposium on Security and Privacy (S&P)*. IEEE, 497–512.
[31] Tielei Wang, Tao Wei, Zhiqiang Lin, and Wei Zou. 2009. IntScope: Automatically Detecting Integer Overflow Vulnerability in X86 Binary Using Symbolic Execution. In *Proc. the Network and Distributed System Security Symposium (NDSS)*.
[32] Zhi Wang, Xuxian Jiang, Weidong Cui, Xinyuan Wang, and Mike Grace. 2009. ReFormat: Automatic Reverse Engineering of Encrypted Messages. In *European Symposium on Research in Computer Security (ESORICS)*. 200–215.
[33] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. 2018. QSYM: A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing. In *Proc. the USENIX Security Symposium (SEC)*. USENIX Association, 745–761.
[34] Michal Zalewski. 2014. American fuzzy lop. http://lcamtuf.coredump.cx/afl/

APPENDICES

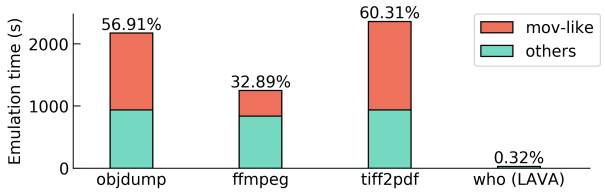


Figure 12: Symbolic emulation time and the proportion incurred by `mov`-like instructions in each program. The percentage on the top of each bar represents the proportion of `mov`-like instructions.

A EMULATION TIME OF MOV-LIKE INSTRUCTIONS

To examine the effectiveness of eliminating `mov`-like instructions with regard to performance improvement, we performed a micro-evaluation. Figure 12 shows the symbolic emulation time taken in each program and particularly the proportion incurred by `mov`-like instructions. In three real-world programs, such as `objdump`, `ffmpeg`[10], and `tiff2pdf`, `mov`-like instructions occupied a great portion of the emulation time. But why? Our observation is that `mov`-like instructions don't introduce new symbolic expressions but they incur a copy of symbolic expressions per execution. At this time, new symbolic expressions are generated for the destination (e.g., memory address or register) of `mov`-like instructions, incurring the performance overhead similar to the introduction of new symbolic expressions. Interestingly, we observed that Qsym occasionally used a solver to concretize memory addresses in the emulation of `mov`-like instructions, incurring the additional overhead.

Unlike the three programs, we observed the small portion of the emulation time for `mov`-like instructions in `who` of LAVA-M. This is because `who` is much smaller in size and less in handling input-related instructions than the other real-world programs. In addition, as shown in Figure 3, `who` doesn't generate complicated constraints, resulting in a fast copy of symbolic expressions.

Our evaluation results indicate that it is obviously effective to remove `mov`-like instructions in advance for performance improvement in hybrid fuzzing.

[10] `ffmpeg` terminated with "Pin is out of memory" error before its complete execution.

Table 16: LAVA-M dataset evaluation results. Left columns: the number of bugs injected in four programs, as listed in LAVA-M [11]. Middle columns: the number of bugs found by recent techniques, as reported in each paper. Right columns: the number of bugs found by VUzzer, Qsym, and Intriguer in our own experiments on the same platform. Our experimental results are reported with max values by 20 runs. Each experiment time is 5h as default [11]. For median values, readers are referred to as Table 8. For extended 24h experiments with who for Qsym and Intriguer, readers are referred to as Figure 8. Note that at present only Intriguer detected all the bugs in four LAVA-M's buggy programs.

Programs	Bugs	FUZZER [11]	SES [11]	VUzzer [27]	Steelix [19]	T-Fuzz [26]	Angora [8]	Qsym [33]	VUzzer	Qsym	**Intriguer**
uniq	28	7	0	27	7	26	28	28	28	28	**28**
base64	44	7	9	17	43	43	44	44	40	44	**44**
md5sum	57	2	0	0	28	49	57	57	26	57	**57**
who	2,136	0	18	50	194	63	1,443	1,238	29	1,452	**2,136**
Total	2,265	16	27	94	272	291	1,572	1,367	123	1,581	**2,265**

Table 17: Time decomposition of field-level constraint solving for performance evaluation of eliminating uncomplicated constraints. For complicated (C) constraints, we used a solver, denoted as S(), but for uncomplicated (U) constraints, we directly solved them as D() or used S() for comparisons. Symbolic emulation time as well as solving time is significantly affected by the uncomplicated constraints. The performance gain by Intriguer, *i.e.*, S(C),D(U) compared to S(C,U), is represented by a percentage. (Time for a single run)

Program	Emulation time (s)			Solving time (s)		
	S(C),D(U)	S(C,U)	Gain	S(C),D(U)	S(C,U)	Gain
md5sum (L)	5.1	7.6	32.9%	1.1	1.7	35.3%
who (L)	2.6	4.3	39.5%	0.6	1.0	40.0%
objdump	2.0	4.2	52.4%	0.7	1.5	53.3%
nm	1.0	2.1	52.4%	0.3	0.7	57.1%
readelf	3.3	6.4	48.4%	1.3	2.3	43.5%
ffmpeg	2.3	3.9	41.0%	0.8	1.3	38.5%
tiff2pdf	4.7	6.2	24.2%	1.7	2.1	19.0%

B PERFORMANCE GAIN BY ELIMINATING UNCOMPLICATED CONSTRAINT SOLVING

To examine the effectiveness of eliminating uncomplicated constraints for a symbolic solver in hybrid fuzzing, we also conducted a micro-evaluation. Table 17 shows that the strategy of Intriguer for uncomplicated constraints (i.e, solving them directly) significantly outperforms the previous strategy (i.e., solving them always with a symbolic solver) in terms of symbolic emulation time and solving time. The performance gain by Intriguer, computed as $(1 - \Delta(S(C), D(C))/\Delta(S(C, U))) \times 100$ for time measurement of $\Delta()$, indicates that eliminating uncomplicated constraints for a solver is an efficient strategy in hybrid fuzzing. Note that compared to solving time, the emulation time more affects performance.

C SPECIAL CASES IN TAINT PROPAGATION

`MUL`, `IMUL`, `DIV` and `IDIV` can affect registers that are not directly used as operands. For example, `MUL` multiplies the value received by the operand by `EAX` and stores the result in `EDX:EAX`, even if `EDX` is not used as the operand. Therefore, when `MUL`, `IMUL`, `DIV`, or `IDIV` is executed, even registers that are not listed in the operand will perform taint propagation.

`XOR` is also used to initialize the operand value to zero in the program. For example, when `XOR EAX, EAX` is executed, the value of `EAX` is always zero. Thus, if `XOR` is used for initialization, the operand used is removed from the tainted memory list. `AND` is also used to mask data in the program. For example, when `AND EAX, 0x000000ff` is executed, only the LSB value is used in `EAX`. Therefore, if `AND` is used to mask data, the data in the unused offset are removed from the tainted memory list. `REP` and related repeat mnemonics are prefixes that can be added to a string instruction for repetition up to the value stored in the counter register. The `REP` prefix is often added to instructions such as `MOVS` to write or read values on memory. If there is a `REP` prefix, the value in the counter register is checked and the result of the repeated instructions is processed immediately. Such cases were also discussed in the previous studies of taint analysis [16, 22].

```
#include <iostream>

class example{
int var1, var2;
public:
void set_request(int a, int b);
int sum(){return (var1+var2);}
}class_obj;

void example::set_request(int a, int b)
{
 var1 = a;
 var2 = b;
}

int main(void)
{
 class_obj.set_request(1,4);
 std::cout<<"\n The sum is "<<class_obj.sum()<<"\n";

 return 0;
}
```

Listing 5: C++ source code used for a binary seed.

D C++ SOURCE CODE FOR BINARY SEED

The source code shown in Listings 5 was used for evaluation of `binutils` programs.

E LAVA-M RESULTS

Table 16 shows the number of bugs found by the state-of-the-art fuzzers in 5h fuzzing on LAVA-M binaries. The middle columns are numbers reported by each paper, while the last three columns are the results of our experiments. Note that we show only the number of "listed" bugs in Table 16 (and also in Table 8) by following previous works, but Intriguer also found "unlisted" bugs, e.g., not only all 2136 listed bugs, but also 328 unlisted bugs in `who`, summing up to 2464 bugs discovered in `who` (5h).