TIFF: Using Input Type Inference To Improve Fuzzing

Vivek Jain
International Institute of Information Technology
Hyderabad, India
vivek425ster@gmail.com

Sanjay Rawat
Vrije Universiteit
Amsterdam, NL
sanjayr@ymail.com

Cristiano Giuffrida
Vrije Universiteit
Amsterdam, NL
giuffrida@cs.vu.nl

Herbert Bos
Vrije Universiteit
Amsterdam, NL
herbertb@cs.vu.nl

ABSTRACT

Developers commonly use fuzzing techniques to hunt down all manner of memory corruption vulnerabilities during the testing phase. Irrespective of the fuzzer, *input mutation* plays a central role in providing adequate code coverage, as well as in triggering bugs. However, each class of memory corruption bugs requires a different trigger condition. While the goal of a fuzzer is to find bugs, most existing fuzzers merely approximate this goal by targeting their mutation strategies toward maximizing code coverage.

In this work, we present a new mutation strategy that maximizes the likelihood of triggering memory-corruption bugs by generating fewer, but better inputs. In particular, our strategy achieves *bug-directed* mutation by inferring the type of the input bytes. To do so, it tags each offset of the input with a basic type (e.g., 32-bit integer, string, array etc.), while deriving mutation rules for specific classes of bugs. We infer types by means of in-memory data-structure identification and dynamic taint analysis, and implement our novel mutation strategy in a fully functional fuzzer which we call TIFF (Type Inference-based Fuzzing Framework). Our evaluation on real-world applications shows that type-based fuzzing triggers bugs much earlier than existing solutions, while maintaining high code coverage. For example, on several real-world applications and libraries (e.g., poppler, mpg123 etc.), we find real bugs (with known CVEs) in almost half of the time and upto an order of magnitude fewer inputs than state-of-the-art fuzzers.

CCS CONCEPTS

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

KEYWORDS

Fuzzing, vulnerability/bug detection, Taint-flow analysis, security, type inference, data-structure Identification

ACSAC '18, December 3–7, 2018, San Juan, PR, USA

ACM ISBN 978-1-4503-6569-7/18/12…$15.00
https://doi.org/10.1145/3274694.3274746

ACM Reference Format:
Vivek Jain, Sanjay Rawat, Cristiano Giuffrida, and Herbert Bos. 2018. TIFF: Using Input Type Inference To Improve Fuzzing. In *2018 Annual Computer Security Applications Conference (ACSAC '18), December 3–7, 2018, San Juan, PR, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3274694.3274746

1 INTRODUCTION

Ever since Barton Miller introduced the concept of fuzzing almost 3 decades ago [35], fuzzing has played a vital role in discovering bugs and evolved from a "*dumb*" (but effective) software testing technique to a range of sophisticated, "*smart*" methods for the systematic security analysis of real-world software [10, 12, 13, 22, 24, 26, 40, 47, 50, 51].

Irrespective of the complexity and nature of the analysis, most modern fuzzers at heart consist of very similar building blocks to implement an *evolutionary* fuzzing strategy. Specifically, at their core, they all contain a component that generates inputs for each new test iteration by mutating the inputs of the previous iterations. Likewise, most fuzzers have a component to assess how well a set of input bytes performs with respect to some objective. The objective of the mutation may differ, depending on the fuzzing strategy. For instance, for *directed fuzzing*, the mutation needs to overcome the challenge of *path diversion*, where the mutation operation should generate inputs that drive the execution of the application towards a *target*, whereas for *coverage-based fuzzing*, the mutation operation should generate a diverse set of inputs to execute as many different paths in the application as possible. As a result, mutation in directed fuzzing is typically more constrained by design. Coverage-based fuzzing, on the other hand, is more welcoming to different mutation strategies while adhering to a set of coverage-oriented heuristics. Of course, the freedom to mutate inputs can easily lead to the generation of many uninteresting or invalid inputs for every interesting one [18], and as a result, there have been several attempts to mutate more *sensibly* in coverage-based fuzzing [10, 13, 40, 47]. However, in this paper we will show that even the most advanced fuzzers still generate a lot of useless inputs and because of this fail to satisfy the specific conditions for triggering a bug. We then demonstrate how knowledge of types helps overcome this issue.

While the ultimate aim of fuzzing is to find bugs, most mutation strategies in coverage-based fuzzers focus on modifying bytes of an input such that the program executes previously unseen code. For example, Driller [47] relies on concolic execution to find and solve branch constraints to get new inputs that execute different

paths. Similarly, VUzzer [40] uses dynamic taint analysis (DTA) to detect bytes ("offsets") in the input that end up as operands in `cmp` instructions and changes them to trigger new paths. However, the mutation of most input bytes still relies on random, input format-agnostic values.

The key insight in this paper is that in the end, in existing solutions it is exactly this *random* mutation that triggers the bug conditions—mostly by brute forcing many bytes. In other words, while the smart mutation strategies in today's fuzzers help achieve good code coverage, they still spend a huge amount of mutation *effort* on triggering a bug by randomly trying, say, hundreds or thousands of bit flips.

In contrast, we propose a new mutation strategy that uses input type inference to address this aspect of input mutation not only for code-coverage maximization, but also for maximizing the *likelihood of triggering memory corruption bugs*. In particular, we show that by inferring types for every offset of the input, we can prioritize not just important *offsets*, but also the *values* at those offsets to improve coverage of both code and bugs (Section 3).

For instance, to achieve good code coverage, we determine which input bytes influence the program's control flow (e.g., end up as operands in `cmp` instructions) and mutate them in accordance with their inferred types. Thus, we modify an 8-bit integer to adhere to their types: the integer may take several interesting values in between 0 and 256. Doing so reduces the number of runs with invalid inputs and covers more code with the same number of inputs. Likewise, for bug detection, the same type inference allows us to mutate certain offsets of the input to trigger certain classes of bugs. For instance, an integer overflow, by definition, involves data of type `INT`. If we can infer that certain bytes in the input are of type `INT`, we can mutate them with *interesting* `INT` values (e.g., a very large integer) to increase the chance of an integer overflow.

To *infer* the types of offsets in the input, we use in-memory data structure identification (DSI) techniques to identify the types of each memory address used by the application, and dynamic taint analysis (DTA) to map what input bytes end up in what memory locations. By combining these two *mappings*, we associate a type with each byte or combination of bytes of the tainted input (Section 4). Our prototype implementation, TIFF, currently builds on two existing dynamic type inference methods [32, 45], but is agnostic to the particular method used and can work with other DSI techniques as well. The types that we consider (infer) in this paper are integers of size 8-, 16- and 32-bits (without inferring signedness) and `struct/arrays` of these basic types. For few cases, we are able to infer signedness of the offsets precisely (thanks to the technique, stated in Reward [32]). As we shall see, by mutating inputs in a type-consistent manner (Section 5), TIFF triggers bugs much earlier than other systems by focusing on the most interesting offsets and values in the input. Ultimately, TIFF shows that type-inference techniques can help reduce the gap between grammar-based generational fuzzers (which are more effective thanks to knowledge of the input format) and modern mutational fuzzers (which can better support arbitrary real-world applications with unknown input format, at the cost of a less efficient fuzzing strategy).

Focusing on common low-level bugs such as integer and buffer overflows, we evaluated our TIFF prototype on two datasets: LAVA-M [18] and MA (miscellaneous applications, which consists of several real-world applications and libraries (see Table 2). Our evaluation shows that type-based mutation triggers bugs an order of magnitude faster than state-of-the-art fuzzers (Section 7).

This paper makes the following contributions:

- We motivate the issue of performing mutation more effectively by finding gaps in the way modern fuzzers perform mutation.
- By applying the existing input reverse-engineering and DSI techniques, we present a new type inference-based mutation strategy that enhances code coverage as well as the probability of triggering memory corruption bugs.
- We implement the proposed technique in a fully functional fuzzer, called TIFF, which will be made open source soon (updates can be found on https://www.vusec.net/projects/#testing).
- We evaluate TIFF on several real-world applications to empirically show its effectiveness.

2 MOTIVATION

In this section, we provide background on evolutionary fuzzers to set the stage for the technique proposed in this paper. Moreover, we evidence the limitations of current state-of-the-art fuzzing techniques by means of a motivating example.

2.1 Evolutionary Fuzzing

Evolutionary fuzzing is a special case of the application of evolutionary algorithms for input generation [30]. Like any evolutionary algorithm, evolutionary fuzzing involves *mutation operators*, *fitness criterion*, and a *feedback loop* to generate newer generations of inputs.

As an example, AFL [51] is a state-of-the-art evolutionary fuzzer that uses genetic algorithms to drive its input generation. In AFL, the fitness criterion for an input is its ability to execute a newer *edge* in the control-flow graph. With its simple fitness criterion and mutation strategy, AFL's feedback loop selects inputs that in one run have discovered new *edges* for the next generation. It should be noted that AFL has no feedback on its mutation strategy, i.e., it does not know where in the input to mutate to maximize the chance of discovering new basic blocks. This causes AFL to *waste* a lot of mutation time on invalid/uninteresting inputs. AFLFast [10] addresses this problem by assigning a *probability* to each input based on how often paths are taken (high- or low-frequency) and uses *power schedules* to select inputs for mutation. However, it still does not solve another common problem for fuzzers, which is how to locate the most appropriate *offsets* in the inputs to mutate. A more recent solution, VUzzer [40], addressed this problem of finding *interesting offsets* for mutation by making use of dynamic taint analysis (DTA).

VUzzer is an evolutionary fuzzer that fuels its evolutionary fuzzing loop by considering *data-* and *control-flow* features of the application being fuzzed. VUzzer selectively applies DTA to check which bytes in the input reach instructions such as `cmp`, which commonly determine branch outcomes. It uses this information to infer the presence of *magic bytes* and *markers* in the input file, which are later used to mutate inputs (thereby reducing brute forcing such values).

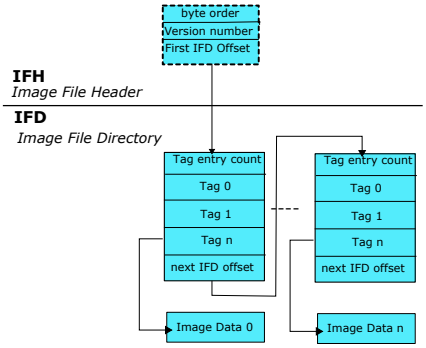

Figure 1: A high-level structure of a tiff file: an 8-byte header is followed by a sequence of image file directory (IFD) structures.

VUzzer applies the expensive DTA technique in a selective manner to find other interesting offsets. While mutating an input, it particularly targets such offsets (applying several mutation operations). By doing so, VUzzer is able to generate valid inputs that traverse different parts of the application quickly.

While VUzzer presents a promising approach to mutate inputs by targeting only interesting offsets in the input, we observe that, apart from detecting magic bytes/markers, it just makes an *educated guess* in mutating other offsets. Such a blind mutation may not be effective to trigger bugs. In order to illustrate these issues in a more concrete manner, we now present a motivating example.

2.2 Motivating Example

To bring forth the key idea behind the proposed technique, we consider an example of an input format (*tiff*)and its processing by an application (*libtiff*).

Figure 1 shows the organization of a tiff file. It has an 8-byte header in which the last 4 bytes determine the position of the image file directory (*IFD*) offsets. The bytes between the *IFD* offset position and the header bytes may or may not be processed by the application, depending on the other tags and the file size. Therefore, determining how the application processes these bytes is crucial to have a meaningful mutation of bytes.

In the *IFD* structure, the first 2 bytes determine the number of 12-byte tags and are followed by specified numbers of such fields. Listing 1 shows an example of vulnerable C code which parses the tiff file format. It is based on the `TIFFRGBAImageBegin()` function of the libtiff library [31], with an artificially injected bug.

In Listing 1, the `ifd_offset` field of `struct header` indicates the start in the file of the image file descriptor (IFD) structures. Each *IFD* structure contains a value that indicates the number of tags present, a list of tags and, optionally, the offset of the next *IFD*. On line 20, the function uses the number of tags to determine the amount of memory to allocate for the 12-byte tag structures. Unfortunately, the `alloc_sz` value can easily overflow if the corresponding byte value is more than 5460. The result is that the buffer overflows when the program tries to copy data in the buffer on line 24 (e.g., causing a segmentation fault).

```
1 typedef struct ifd {
2    uint16_t no_entries;
3    tag* taglist; /* pointer to 12-byte tag structure */
4    int32_t next_ifd_offset;
5 } ifd;
6
7 // Pre: file contents in fbuf, file size in fsize
8 int tiffImageBegin (char* fbuf, uint32_t fsize) {
9    struct header {
10   char identifier[4];
11   uint32_t ifd_offset;
12   } h;
13   memcpy(h.identifier, fbuf, 4);
14   if(memcmp( "II*\0", h.identifier, 4) == 0) {
15      ifd i1;
16      memcpy(&h.ifd_offset, fbuf+4, 4);
17      if(h.ifd_offset + sizeof(uint16_t)> fsize )
18      exit(0);
19      memcpy(&i1.no_entries, fbuf+h.ifd_offset, 2);
20      uint16_t alloc_sz=12*i1.no_entries; ← BUG: overflow!
21      if(h.ifd_offset+2+alloc_sz > fsize)
22      exit(0);
23      i1.taglist = malloc(alloc_sz);
24      memcpy(i1.taglist,fbuf+h.ifd_offset+2,12*i1.no_entries);
25      } else exit(0);
26 ...
27 }
```

Listing 1: Motivating example that illustrates issues in existing fuzzers

Although the bug trivially depends on specific bytes of the input file, it is very hard for general-purpose fuzzers, such as VUzzer [40] or AFL [51], to mutate the input at those bytes and trigger the bug. Specifically, when we ran this (trivial) code snippet with VUzzer, it took as many as 5,000 inputs for VUzzer to crash the application. In contrast, TIFF produced a crash in just 200 inputs. In the following, we explain the reason behind this.

(1) Since the `cmp` for line 17 uses `h.ifd_offset` as its operand, fuzzers such as VUzzer will mutate the value of `h.ifd_offset` and in doing so change the position of the first ifd offset. In contrast, TIFF only changes the intended offset value to try and trigger the bug and this enables it to produce a crash in the given example much sooner.

(2) To trigger the integer overflow on line 20, the fuzzer needs to pick a suitable value for `i1.no_entries`, so that `alloc_sz` becomes too small and a heap overflow occurs on line 24. Existing fuzzers simply try to mutate these bytes in the input in a random way. In contrast, TIFF is aware of the type of `i1.no_entries` and quickly triggers the bug by choosing interesting `INT16` values (that may cause integer overflows).

2.3 Lessons learned

In the light of the above example, it becomes clear that: (i) tailoring the mutation to some interesting values to trigger specific vulnerabilities may boost the fuzzing process, (ii) we can increase the probability of triggering these specific vulnerabilities if we know the types of these offsets, (iii) mutating at every offset of the input may not produce any interesting input, and (iv) mutating at an offset with random values may not produce any interesting input either.

As we can note, in general, there may be several offsets that are used by the application, and many of them are used in a *sensitive* way. If we target these offsets and mutate them according to the way these are *used* by the application, we may perform mutations more

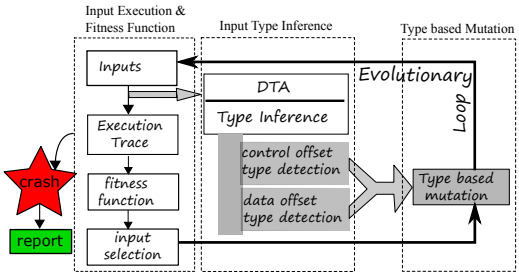

Figure 2: High-level overview of TIFF. Each solid block represents a different component, while each dashed box represents a different high-level functionality of the fuzzer.

efficiently. Therefore, to understand *how certain offsets are used by the application*, we *infer* the types of those offsets. Subsequently, we mutate the data at those offsets according to their types. In the next sections, we provide details on our proposed technique.

3 OVERVIEW

TIFF is based on the concept of evolutionary fuzzing and a full fuzzing cycle therefore consists of a sequence of steps. The cycle begins with executing the application on a set of inputs. From the execution traces, evolutionary fuzzers may extract interesting information about the program. In our case, for instance, we extract information about the flow and types of data. Fuzzers then determine the fitness of the inputs using a fuzzer-specific criteria, and select the fittest inputs as promising starting points for the next generation. Subsequently, they will mutate the promising inputs, for instance by flipping bits or inserting bytes. A key advantage of TIFF is that it will do the mutation on the most promising bytes as *sensibly* as possible—taking into consideration collected properties such as type information. Fig. 2 presents the main components of TIFF as well as the interaction among them. The dashed boxes indicate the division of tasks which we now explain in turn.

3.1 Input Execution and Fitness Function

As a mutation-based fuzzer, TIFF needs a set of seed inputs to start fuzzing. The application executes these inputs and produces an execution trace. In the current implementation, TIFF monitors basic-blocks and their execution frequency and calculates the fitness of an input on the basis of the executed basic-blocks. Any input that executes a new basic block is considered for further mutation.

3.2 DTA and Input Type Inference

Dynamic taint analysis (DTA) plays a central role in determining several interesting properties of the input. To maximize the code-coverage and bug detection, TIFF derives two classes of features: *control offset types* and *data offset types*

Control offsets indicate the bytes in the input that influence the operands in `cmp` instructions and determine the outcome of branch instructions. Note that like VUzzer, TIFF also performs DTA while executing an input to find `cmp` instructions whose operands are tainted by some offsets of the input. Such offsets are interesting targets for mutation to change the execution path of the application. TIFF further analyzes this information to infer *invariants* that the application expects from the input. These invariants, such as the presence of *magic-bytes, markers* are widely prevalent in binary input formats. TIFF also computes the types of such offsets by performing a separate analysis for type inference (Section 4) and accordingly associates type tags (such as `INT8`, `INT16`, `UINT32` and `char*`) with these offsets.

Besides control offsets, TIFF performs the type inference technique (Section 4) to associate a type tag with other offsets of the input. We refer to them as data offset types. Currently, TIFF associates `INT8, INT16, INT32` and `array/struct` types to data offsets.

3.3 Type Based Mutation

This is the main step that is responsible for mutating inputs towards high code-coverage and bug detection. For a given input, TIFF first considers the control offset types. It mutates the corresponding offsets either with the *invariants* learned for these offsets, or according to the *type* tag associated with this offset, in case there is no invariant associated with these offsets. Both options improve the fuzzer's *code coverage*. Next, TIFF considers the data-offset types for non control offsets of the input. Here, it performs type-based mutation selectively—on selected inputs that cover a new path only. The intuition is that by focusing on data-offsets, we explore bugs that may lie in this execution path. TIFF' s mutation strategy differs depending on the type of the input bytes. Specifically, for offsets of type `INTx`, TIFF finds *unusual* (e.g., extreme values for a given integer type) values based on the size `x` and places those values at those offsets. This type of mutation mainly targets *integer-overflows* bugs and (to a lesser extent) *heap-overflow* bugs. For offsets of type array, TIFF inserts data (based on the array element type) of arbitrary length. This type of mutation mainly targets *buffer-overflow*

4 INPUT TYPE INFERENCE

In the literature, there exist several type inference techniques, each with their own strengths and weaknesses [11, 16, 32, 33, 38, 45]. Given the nature of our application, fuzzing, we want an algorithm that is fast enough to work on multiple inputs, while providing type information that is sufficiently precise for our task. Unlike other application domains (such as binary rewriting [46]), fuzzing can suffer *some* imprecision in type identification as misclassifications merely lead to a reduction in fuzzing efficiency. For this purpose, we developed a custom technique that, as we shall see, builds on Tupni's input format inference [16], Howard's data structure extraction based on memory accesses [45], and REWARD's data structure identification based on known API calls [32], but addresses key challenges when complementing such techniques in a unified, practical type identification system to boost fuzzing.

4.1 In-memory Data Structure Identification for Input Offsets

As mentioned earlier, to mutate more effectively, we need to learn the *type system* on the input. As TIFF mainly needs to cater to binary input formats (TIFF focuses on applications that consume binary files), techniques for learning grammars may not work well [19]. Binary files are often organized as arrays of data types such as

long and short integers, chars and strings. Our goal, therefore, is to learn this type system automatically. More precisely we want to understand how the application processes each offset of the input. We identify the following two categories of data types associated with input offsets: (i)- individual *n*bytes values (e.g., 1byte, 2byte, 4byte, etc.), (ii) composite bytes (i.e., a set of offsets which are processed as an array or struct).

Our in-memory DSI step consists of three components: basic data type identification, composite data type (e.g., array) detection and, precise detection of certain data types such as char*, int, etc. For a given input i, the outcome of this step is a mapping $\psi : i[] \rightarrow T$, where $i[]$ is a set of all offsets of the input i and $T = [INT8, INT16, INT32, array/struct]$. T denotes the types that are recognized by TIFF. To explain with an example, if we get $\psi(i(2)) = INT8$, it means that the 3^{rd} offset of the input i is of type $INT8$. To support such type detection, TIFF employs a DTA engine to monitor the flow of tainted inputs within the application.

DTA determines, during program execution, which memory locations and registers are dependent on tainted input bytes. Based on the granularity, DTA then traces back the tainted values to individual offsets in the input. Our DTA framework is based on LibDFT [28].

4.2 Basic Data Type Identification

Using Tupni's technique of input format inference [16], we identify types associated with (a set of) offsets in the input, based on the observation that an application processes offsets almost exclusively based on their type information. In other words, it processes a 4-byte data item in the input (which could be of type INT32) as a chunk of 4 bytes in the application logic.

In short, Tupni's algorithms works as follows: we partition the input into short sequences of consecutive bytes and monitor the application to know how instructions are accessing the tainted bytes. For example, consider an `add` instruction such as `add reg32, [addr]` where `[addr+0, addr+1, addr+2, addr+3]` are tainted by file offsets 0, 1, 2, and 3 respectively. In this case, we classify the 0th byte as a chunk of size 4. We also assign a weight to each chunk where the weight indicates how many times that chunk has been accessed. We notice that the chunks may not always be disjoint. For all pairs of intersecting chunks, we retain the chunk with the higher weight.

4.3 Array Detection

For composite data types such as arrays and structs, we use Howard's in-memory array detection [45]. We choose Howard's array detection technique, as it is more precise and overcomes most of the limitations of other techniques (such as those of Tupni which draw inspiration from Polyglot [11]). Howard is a dynamic analysis technique to recover data structures present in a binary.

Howard first identifies root pointers that are not derived from any other pointers. It then identifies base pointers dynamically by tracking the way in which the program derives new pointers from existing ones, and how it dereferences them.

On top of Howard's techniques, we associate another tag with each memory address and general purpose register to record whether the addresses/registers are tainted by any offset of the input. Thus, whenever Howard detects an array, we check whether the memory in the array is tainted by offsets of the input, thereby recovering all the offsets of the input which the application processes as an array. We observed that in some cases, because of the limitations of Howard, some memory locations which are part of an array, are not recovered as tainted. We apply heuristics to solve these cases. A typical example is given below:

```
1  struct header {
2  int len; /* total length of struct */
3  char identifier[];
4  } *element;
5  /* ... assign some value to element ... */
6  for (int i = 0; i < element->len; i++)
7  printf("%02x ", ((unsigned char*)element)[i]);
```

Listing 2: A problematic case for array element detection in Howard [45]

While not common, a program may access array elements with respect to the start of a struct rather than the start of the array. Listing 2 shows an example. In this case, Howard also classifies `len` as an element of the array `identifier`. To filter such cases, we verify if the difference in the addresses of successive memory locations of the array remains constant. For example, let's say integer `len` in line 2 has address $a1$. In that case, the array elements in line 3 will have addresses $a1 + 4$, $a1 + 5$, $a1 + 6$ etc. We now eliminate `len` from being an element of the array since the difference in the address of `len` and that of the first array elements is not consistent with the difference between the addresses of the other array elements. Clearly, this is not a very strong heuristic and it would would fail in cases where the types are the same, but this is good enough in practice; fuzzing can easily tolerate some imprecision in type inference. We provide few more finer details of our engineering efforts on top of Howard's original implementation in Appendix 9.2.

4.4 Precise Data Type Identification

Finally, we also use a limited version of REWARD [32] to identify more precise data types, such as `size_t (unsigned int)`, `char*`, etc. We achieve this by hooking *libc* library calls for which we have detailed type information for the arguments. For example, for library calls `strcpy` or `strcmp`, we know that the arguments are of type `char*`. Thus, using our dynamic taint analysis we check if the arguments of such library calls are tainted by any offset in the input. Additionally, for some of the string comparison APIs such as `strcmp`, `strncmp`, `memcmp`, we also record the input offsets, as well as the bytes to which they are compared. We use these offsets and bytes later in our mutation strategy to increase coverage. For the current implementation of TIFF, we hook 17 library calls from libc (for example, inferring `char *` from `strlen, strnlen, strdup, memchr, ...` and `size_t` from `malloc, strndup, memcpy, memchr, ...`).

5 TYPE INFERENCE-ASSISTED MUTATION

After obtaining the type information for the input in the form of the mapping ψ defined in Section 4.1, TIFF uses it to mutate inputs to achieve the goal of high code coverage and early bug detection[1]. Specifically, since we know the data type of the input offsets, we can mutate these offsets more *meaningfully*. In the following, we discuss

[1]Thanks to the input type inference assisted value selection for mutation, keeping a particular type of memory corruption bug in mind

how our mutation strategy helps achieving the goals of coverage-oriented and bug-oriented mutation. An algorithmic description is provided in Appendix 9.1.

5.1 Coverage-oriented Mutation

TIFF achieves our goal of high code-coverage by taking advantage of the type of offsets that correspond to the operands of `cmp` instructions. Although a solution such as VUzzer also detects the offsets of `cmp` instructions, it is unaware of the data type of these `cmp` offsets. As a result, it *wastes* a considerable amount of its mutation effort on mutating at such offsets with arbitrary values. For example, if the offset used in a `cmp` instruction is of type `INT8` (i.e., a byte), we have 2^8 different values to choose from for mutation. However, VUzzer commonly tries to mutate it (and surrounding offsets) by interpreting it as part of a `INT32` type, using values from the set of 2^{32} possible integers. In case of TIFF, if the offset 0 is of type `INT32`, TIFF would mutate these 4 bytes together, instead of mutating only a single byte.

We also improve code coverage by replacing the input bytes at an offset with the bytes which we have recorded by hooking the *string-compare* family of library functions. For example, for a function call with `memcmp("II*", a)` (as in our motivating example in Section 2), where `a` is tainted by offset `0` during execution on all the mutated inputs, we replace the bytes at offset `0` with the string `"II*"`. These features of TIFF in generating valid inputs are more effective than those in existing fuzzers such as VUzzer because in some cases VUzzer will miss these strings. For example, in Listing 3 VUzzer was unable to get the byte `%` for offset `0` of the file in `cmp.out`. When we further analyzed the issue, we observed that internally, in the assembly of `memcmp`, if the string that is compared has a size of 5 bytes, the first byte value is first taken into a register and then that value is subtracted from the tainted value. If the subtracted value is 0 (i.e., they are equal), it takes a jump to the true branch (i.e., to a basic block that does another `cmp` with the rest of the 4 bytes). Otherwise, it jumps to the false branch. Thus, since there is no `cmp` for the first byte, VUzzer misses the magic byte altogether. In contrast, because of TIFF's hooking and recording of such tainted library functions, it is able to detect these bytes precisely and generate valid inputs accordingly.

```
1 if (memcmp(buf,"%PDF-",5)==0) // buf tainted by offsets 0-5
2 do_something();
```

Listing 3: Comparing bytes with memcmp. Missed when monitoring `cmp` only

5.2 Bug-oriented Mutation

This type of mutation mainly targets the offsets of type *data offsets*. In other words, while selecting the offsets for mutating the input to generate next input, we consider offsets of a particular data type, along with the offsets that are used in any control-flow decision. In the current implementation, we specifically increase the probability of detecting two classes of memory corruption bugs: integer overflows and buffer overflows. Integer overflow bugs occur when an integer exceeds its maximum value or in the case of bad casting between the types of the variables involved in some assignment, such as the interpretation of a `signed` variable as an `unsigned` one. Buffer overflow bugs occur when the amount of data copied into a memory buffer exceeds its size.

To increase the probability of integer overflow bugs, TIFF periodically chooses an input that contains the highest number of offsets with type `INTx`. The period is a (configurable) parameter n whose value can be configured on the basis of the size of the seed inputs. In our experiments, we found $n = 10$ to be a good value for binary inputs. For the chosen input, when the fuzzer encounters offsets with types such as `INT16` or `INT32`, it will modify them using interesting integer values—for example, the values used by AFL [52].

Similarly, to increase the probability of triggering buffer overflow bugs, we choose the input which has the highest number of offsets associated with type `array` and then try to increase the size of these arrays by inserting byte strings of some arbitrarily chosen length. The place where we add the additional sequence of bytes is chosen randomly between the array starting offset and ending offset.

For efficiency, we run these bug-oriented inputs without any monitoring or instrumentation. In other words, we do not calculate the fitness value for these inputs. If any of such input results in a crash, we consider the input for mutation in the following generations to produce more inputs. This strategy is an optimization to increase the input execution rate in a given period of time. This optimization is based on the observation that, as the number of *data offsets* is much higher than that of *control offsets*, we end up mutating mainly *data offsets*, thereby reducing the likelihood of executing a new path. However, it should be noted that for *jump table*-based implementations that depend on some non-control tainted input bytes, we may neglect inputs that trigger newer paths on such jumps. But, as noted in [15], common jump table implementations do rely on `cmp` instructions and, if so, our mutation strategy is unaffected.

6 IMPLEMENTATION

This section begins with a discussion of the implementation aspects of TIFF. This also highlights some auxiliary contributions—mainly optimizations in the systems that we used for implementing TIFF.

We build TIFF on top of the open-source fuzzer VUzzer [39]. We chose VUzzer since it is a state-of-the-art evolutionary fuzzer that implements an already efficient coverage-oriented fuzzing strategy (and thus a harder-to-improve baseline).

As part of our implementation, we re-engineered libDFT to make it compatible with 64-bit applications and lifted VUzzer to work on 64-bit systems. We use VUzzer's fitness function. However, we completely reworked VUzzer's mutation strategy to reflect the type inference-based techniques proposed in this paper.

As discussed earlier, part of our input type inference system is based on Howard. To make Howard suit TIFF's purpose, we modified it in several ways, for example, by lifting it to work on 64-bit binaries, by implementing a different data-structure for taintmap that scales well on larger inputs etc.

Finally, in our implementation, we observed that Howard's array detection takes a very long time for some large and complex applications. To achieve faster input generation, we therefore run the array detection only for the seed inputs.

Crash Triage: For comparison purposes, to identify the uniqueness of crashes we use the stack hash technique, described in [36]. Using Pintool [34], TIFF monitors a short execution history upto

the crash point to compute the stack hash. It keeps track of the last 10 executed basic blocks and the last 5 executed function calls in a ring buffer before the crash point and then the hash of the buffer is calculated to determine the uniqueness of the crash. The idea of considering only a short sequence of basic blocks before the crash happens is inspired by the observation made by Arulraj *et al.* [8] that the "short-term memory" of an execution is sufficient for failure diagnosis.

7 EVALUATION

In this section, we evaluate TIFF on several applications. We present results for applications fuzzed for 12 hours[2].

For each application, we gather 3-4 *random, but valid* inputs and we use this as a seed set of inputs for each fuzzer considered in our evaluation. To compare the performance of TIFF against the state of the art, we also present experimental results for (64-bit) VUzzer [40] and AFLFast [10]. We consider the performance in terms of speed (how many unique crashes—a proxy metric for bugs—detected in how much time?). For most of these proxy metric across different applications (that we chose to evaluate TIFF with), we show the overall performance of TIFF by computing geometric means across all the runs. As the values of these proxy-metrics are skewed, arithmetic mean may not be a good candidate to access the central tendency [37].

For our evaluation, we consider two datasets, drawing from recent work in the area [10, 40]. First, we use a set of *buggy* binaries recently generated by the LAVA team [18], specifically the LAVA-M dataset. Second, we consider miscellaneous real-world applications which process binary input data, such as image processing applications (see Section 7.2). We refer to this miscellaneous application dataset as the MA dataset.

Apart from the above mentioned datasets, we separately ran TIFF on the latest version of two applications- `libming-0.4.8` and `libexiv2` 0.27- and found new bugs. In `libexiv2`, we found couple of infinite loops bugs, for example, in function `Exiv2::Image::printIFDStructure()`. We also found few *assertion failure* errors, for example, in function `Exiv2:RafImage::readMetadata()`. In `libming`, we found a access violation in the function `parseABC_NS_SET_INFO`, resulting in a segmentation fault. These issues are reported to the respective vendors.

We ran all our experiments on an Ubuntu 14.04 LTS system equipped with a 64-bit 2-core Intel CPU and 16 GB RAM. Following the recommendations, made by Klees *et al.* [29], we repeated all our experiments 3 times and report the average, with marginal statistical variations observed across repeated fuzzing runs.

7.1 LAVA-M Dataset

In a recent paper, Dolan-Gavitt *et.al.* [18] developed a technique to inject hard-to-reach software faults and created buggy versions of a few Linux utilities for testing fuzzing- and symbolic execution-based bug finding solutions. This dataset consists of 4 Linux utilities

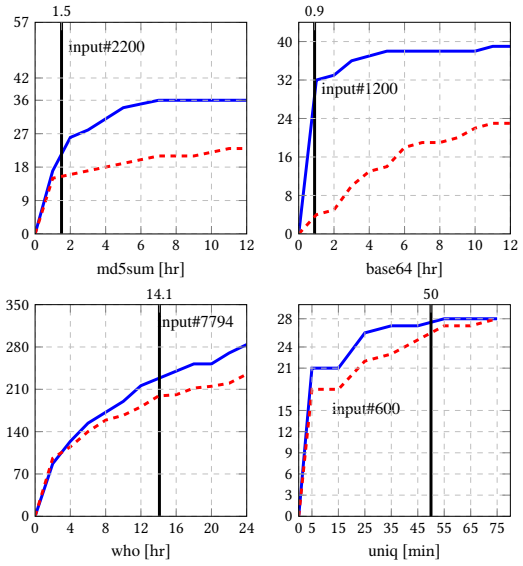

Figure 3: Distribution of crashes for applications over their run time period. X-axis: time. Y-axis: cumulative sum of unique bugs. Blue line: TIFF. Red dashed line: VUzzer. Vertical black line: Time taken by TIFF to find the same number of crashes as those found by VUzzer during a complete run.

`base64`, `who`[3], `uniq`, and `md5sum`. Each of these binaries is injected with multiple faults (in the same binary for each utility). We use this dataset to compare TIFF's performance to that of VUzzer, which has shown good results on the LAVA-M dataset [40]. We also ran AFLFast on this dataset, but AFLFast could not find any bug in the binaries included in the LAVA-M dataset, except for `md5sum`. In the latter case, it reported 2 crashes, which did not match any of the injected faults[4]. We show how TIFF's type-assisted mutation can greatly increase bug coverage on the LAVA-M dataset.

Table 1 presents our results. Each injected fault in the LAVA binaries has a fault ID that is printed on standard output before the binary crashes due to that fault. This allows us to precisely identify the unique bugs triggered by TIFF from the crash runs.

As shown in Table 1, TIFF found more bugs with fewer inputs compared to VUzzer. Moreover, Figure 3 illustrates the distribution of crashes on the LAVA-M binaries over their running period. The y-axis of each plot shows the cumulative sum of crashes and the x-axis of the plot shows the total execution time of the fuzzers. As shown in the figure, not only does TIFF find more bugs than VUzzer, but also finds them sooner.

As TIFF performs mutation in a controlled manner (i.e., the mutation of control offsets and the mutation of data offsets are done in separate cycles), we could also measure the effect of these mutation strategies on TIFF's behavior. We observed that several of LAVA-M fault IDs (each bug has a unique ID in LAVA-M dataset) that TIFF

[2] As we compare TIFF with VUzzer, instead of running each application for 24hrs, as done in VUzzer paper [40], we ran each experiment for 12hrs. In this way, we want to show that TIFF is effective in finding bugs in considerably less amount of time.

[3] Since `[who]` has a large number of bugs, which are difficult to detect in 12 hours, we ran the fuzzers longer (24 hours).

[4] We ran AFLFast with default configurations

Table 1: LAVA-M dataset: TIFF vs. VUzzer. Column 3, 4 and 5 show data as *#unique bugs (total inputs)*. The numbers in brackets show the number of inputs required to generate the unique bugs.

Program	Total bugs	TIFF	VUzzer	AFLFast
`uniq`	28	28 (700)	28(1400)	0 (783k)
`base64`	44	39 (15.4k)	23 (21.6k)	0 (14M)
`md5sum`	57	36 (10.5k)	23 (13k)	2(495k)
`who`	2136	284 (11k)	235 (12k)	0(19M)
geo mean	-	57.8 (5.9K)	43.2 (8.2K)	0.31 (3.1M)

found using mutation of only data offset types, were not found by VUzzer.

7.2 MA Dataset

We now consider our MA dataset of real-world programs to evaluate the performance of TIFF. This dataset consists of the following utilities: jbig2dec (0.11), potrace (1.11-2), pdf2svg (0.2.2-1), gif2png (2.5.8-1), mpg321 (0.3.2-1.1), tcptrace (6.6.7-4.1), tcpdump (4.5.0), djpeg (1.3.0), autotrace (0.31.1-16), pdftocairo (1.13.0) and convert (8:6.7.7.10). We selected these applications by considering experimental results reported in [10, 21, 40]. We did not consider any `binutils` utilities as TIFF is not suitable for applications that involve heavy parsing. For each of these programs, we use their vanilla release for Ubuntu 14.04. By evaluating these utilities, we also targeted some well-known libraries that are used by these utilities, such as libpotrace (1.11-2), libjbig2dec (0.115), libpoppler (0.24.5-2), libpng (1.2.50-1), libasound (1.0.27.2-3), libpcap (1.5.3-2), libautotrace (0.31.1-16), libcairo (1.13.0), libjpeg-turbo (1.3.0) and libMagickCore (8:6.7.7.10)[5].

In order to gather insights into the performance of TIFF, we also ran VUzzer and AFLFast on these applications. For our performance comparison, we chose AFLFast over AFL, as AFLFast has been shown to perform better than AFL [10], but both these fuzzers follow a similar coverage-oriented fuzzing strategy. Table 2 presents our results on the MA dataset.

Since both VUzzer and AFLFast employ a different technique for determining crash uniqueness, in order to have a meaningful comparison of (unique) crashes reported by TIFF, we run the same call stack-hashing based algorithm on the crashes reported by VUzzer and AFLFast for each application. This simple algorithm provides a common uniqueness metric for crash reporting across all the fuzzers.

As shown in Table 2, TIFF again triggers more crashes than VUzzer with generally fewer inputs. This confirms our observation about the type-agnostic mutation performed by VUzzer: since VUzzer does not know the type of the offsets, it is unable to meaningfully mutate input bytes at those offsets. The delta with AFLFast is more pronounced, as TIFF was able to produce many more crashes using an order of magnitude fewer inputs.

This experiment on real-world applications particularly indicates towards an important property to reason over fuzzing effectiveness. Fuzzers such as AFLFast (and AFL) rely on a very lightweight instrumentation, which allows them to process many more inputs (millions) than the inputs process by TIFF and VUzzer's complex binary instrumentation for a given testing time interval (12 hours). However, the higher-quality input mutation strategy produced by such complex instrumentation more than compensates for the slower input processing time, ultimately resulting in more crashes being detected. Even when comparing TIFF and VUzzer, our results show that TIFF's more complex instrumentation does generally result in less inputs processed per time unit, but this still translated to a more effective fuzzing strategy.

We also evaluated TIFF's effectiveness from a code-coverage perspective (another common but less precise proxy metric for bug detection effectiveness). Table 3 presents our results for TIFF and VUzzer. Note that since we fuzz application *binaries*, it is not trivial to simply express code coverage in terms of percentage of the total code. For this reason, we measured the number of new basic blocks covered by the fuzzers over the number of basic blocks that were executed by the fuzzers with seed inputs. Column 2 of Table 3 lists the number of initial (seed input) basic blocks, while the other columns present the number of basic blocks discovered by each fuzzer during the entire testing time interval (12hrs). This number includes only the main application basic blocks and the basic blocks of the libraries that we target. The numbers inside the brackets in columns 3 and 4 indicate the total number of inputs that were executed by each fuzzer. In the case of TIFF, this number also includes inputs generated using data-offsets based mutation (with no monitoring of executed basic-blocks).

As shown in Table 3, for most applications, TIFF is able to discover more basic blocks than VUzzer with fewer inputs. This confirms our assumption that type-consistent mutation of the (now known) type of `cmp` offsets leads to a faster discovery of newer basic blocks. For `mpg321` and `gif2png`, our results only reveal a small (or no) difference between the number of basic blocks covered by TIFF and VUzzer. A closer inspection revealed that, in these cases, the number of inputs generated during the data-offset mutation phase did not produce any crashes and such inputs are not monitored for new basic blocks, but TIFF spent a lot of time on such inputs. This behavior prompted us to further explore the issue of *missing* code coverage. We ran 3 applications (gif2png, mpg321 and autotrace) by enabling the monitoring for bug oriented cycle. We find that we missed 0% (resp. 23%, 26%) basic blocks for mpg321 (resp.gif2png and autotrace).It is obvious that we need a way to capture such new basic blocks, with less execution penalty. Therefore, in the Table 3, we can observe that for certain number of applications, TIFF is not significantly better than VUzzer.

7.3 Crash Analysis

To identify the severity of crashes (and resulting bugs), we examined the crashes discovered in the various applications using `!Exploitable` [20]—a tool developed on top of GDB that classifies bugs by severity and recently ported to crash processing utilities for AFL [41]. `!Exploitable` uses heuristics to assess the exploitability of a crash inside a given application. While by no means a perfect assessment, an indication of exploitability indicates that a bug is serious.

We find that TIFF could trigger *exploitable* bugs in several applications from MA dataset.

[5]It should be noted that several of these applications are also used in original VUzzer paper and we added more applications for the current experimentation.

Table 2: MA dataset: TIFF vs. VUzzer vs AFLFast

Application	TIFF		VUzzer		AFLFast	
	#Unique crashes	#Inputs	#Unique crashes	#Inputs	#Unique crashes	#Inputs
mpg321+libasound	3.33	11.9k	4	14.8k	1	1.4M
pdf2svg+libpoppler	1.66	7.6k	0	4.4k	0	1.6M
jbig2dec+libjbig2dec	32	11.8k	0	12.4k	8	1.7M
potrace+libpotrace	9.33	8.4k	6	11k	7	9M
gif2png+libpng	6	10.6k	9	8.4k	8	13M
tcptrace+libpcap	3	6.1k	4	10k	2	1.8M
autotrace+libautotrace	11	2.5k	9	2.8k	-	-*
pdftocairo+libcairo	2	11.2k	1	8.8k	1	347k
convert(gif)+libGraphicsMagick	1	4.4k	1	2.4k	0	829k
geo mean	4.42	7.5k	2.45	7k	1.70	1.9M

* We could not run AFLFast on this binary.
† We did not mention results for `djpeg+libjpeg` and `tcpdump+libpcap` as we could not find any crash with any of the three fuzzers, evaluated in this experiment.

Table 3: Basic blocks discovered by TIFF and VUzzer on MA dataset.

Program	Initial #BBs	TIFF (#inputs)	VUzzer (#inputs)
`mpg321`	460	597(7400)	597(14800)
`pdf2svg`	4767	5656(3660)	5078(4600)
`jbig2dec`	974	1368(8454)	1076(12400)
`potrace`	1390	1542(2819)	1520(11000)
`gif2png`	1170	1282(7200)	1309(8600)
`tcptrace`	1290	1637(3406)	1405(10800)
`autotrace`	1521	1676(4380)	1604(2800)
`pdftocairo`	4742	4872(6028)	4758(8800)
`convert`	3399	5562(1600)	5480(2400)
geo mean	-	1935.6 (4438.6)	2063.7 (7184.5)

Table 4: Type of bugs discovered by TIFF

Program	Bug Information
`mpg321`	heap overflow
`pdf2svg`	buffer-overflow
`jbig2dec`	jump target corruption; arbitrary write access violation
`potrace`	heap overflow; arbitrary write access violation
`gif2png`	arbitrary read access violation
`tcptrace`	arbitrary write access violation; NULL pointer dereferencing
`autotrace`	arbitrary read/write access violation; buffer-overflow
`pdftocairo`	buffer-overflow
`convert`	buffer-overflow

Finally, on these crash triggering inputs, we also analysed the impact of our mutation strategy. Specifically, we observed that the inputs that triggered the crashes in jbig2dec, pdf2svg, and potrace are generated as a part of data offset based mutation. More precisely, for jbig2dec and potrace, these inputs were generated by targeting offsets of type `int`, thereby causing integer overflow bugs in these applications. We run the crashes found by jbig2dec on it's latest version (0.13) to check the effectiveness of TIFF. We found that on the latest version, application exited by printing "Integer Overflow multiplication from stride(268435456)*height(701)." This shows that TIFF is able to triger bug on the previous version because of it's Type Based Mutation Strategy. TIFF aware of the type of `stride` is able to put special `INT32` values of stride, thus leading to a crash. Similary we ran the crash found by potrace on one of known parser of BMP file `bmp2tiff`. It exited by printing "Cannot process BMP file with bit count 264". With the help of this statement we can identify that TIFF is able to trigger crash on potrace since it was aware of type of the input offset. TIFF mutated the value at that offset with 2byte integer 264. Potrace application has not handled this case, therefore TIFF is able to trigger crash in the application. For pdf2svg, buffer overflow bugs were caused by targeting offsets of type `array`.

Overall, we find that TIFF type-consistent fuzzing of both control and data offsets finds bugs quickly, that both control offset and data offset mutation matters, and, moreover, that some of the bugs we found are very severe, as confirmed by manual inspection.

In particular, based on our crash analysis, we found that TIFF discovered a previously reported CVEs on potrace [1] and autotrace [2, 7]. In jbig2dec, TIFF found integer overflow bugs, one of which has been already reported [3]. Other bugs have been previously reported by VUzzer [4–6].

In Table 4, we report causes that resulted in crashes.

8 RELATED WORK

In this section, we walk over the literature on fuzzing to highlight the contribution made by TIFF compared to existing approaches.

8.1 Directed Fuzzing Approaches

Directed fuzzing, intuitively, can be seen as a way to verify if a seemingly suspicious code could indeed be vulnerable. Some of the existing approaches use some form of symbolic execution to drive the inputs towards the target [23, 26]. In [9], Böhme *et al.* proposed DGF which involves LLVM-based static analysis to find functions and basic-blocks that lead to a set of target error-prone code. While the suspicious code, for example calls to known vulnerable functions or patched code, is often known *a priori*, there are exceptions such as Dowser [26], which implements a symbolic execution-based approach to automatically find code prone to buffer overflows. In contrast, TIFF relies on a bug-oriented mutation strategy to target buffer-overflow bugs, without knowing them a priori in the application or using code-driven heuristics to reduce the (huge) search space. TIFF's taintflow based type inference, together with DGF, proposed in [9], may be used to effectively mutate bytes that influence the branches only on the directed path, thereby driving the execution towards the target faster.

BuzzFuzz [22] is another example of fuzzerthat uses DTA, but on the source code. TIFF also uses DTA to find interesting offsets in the input, but uses this information for code and bug coverage by learning the input properties based on the application behavior. This makes TIFF a more generic fuzzer than directed fuzzing approaches. Moreover, many of these approaches also require the availability of source code to perform analysis, whereas TIFF is able to fuzz binaries of the applications.

In [32], Zhiqiang *et al.* also showed a possible application of REWARD's analysis to directed fuzzing. In this case also, TIFF is different as it does not rely on any vulnerability specific information and it has its own input-driven heuristics to mutate and trigger bugs (in addition to its coverage-oriented strategy).

8.2 Input Grammar-Based Fuzzing Approaches

Grammar-based fuzzing technique is an instance of *generational* fuzzing, wherein the format of the input is known a priori. Such approaches are more effective in fuzzing as by design, as the chances of creating *invalid* inputs are much less. However, availability of input formats (specifications) and a guaranteed *correct* implementation of it are difficult to meet in practice. As a result, this line of research is confined to a class of *highly-structured* input formats, such as scripting languages (JavaScript, perl, etc.), mark-up languages (HTML, XML etc.), where the grammar is available. As examples, IFuzzer [48], LangFuzz [27] and the recently published Skyfire [49] are fuzzers that target JavaScript interpreters and XML type languages.

Very recently, there have also been efforts to learn input grammars automatically and use that knowledge to fuzz [25]. TIFF differs from such approaches in a number of ways. In certain input formats, such as image file formats, the type information is not captured by learning the grammar and hence, the coverage-based fuzzing may not gain much as far as bug detection is concerned. also, most of the fuzzers in this direction have shown a limited learning capabilities for an arbitrary format. Nevertheless, learning a grammar automatically and integrating it with TIFF's type inference-based fuzzing could be an attractive future direction to explore.

8.3 Evolutionary Fuzzing Approaches

Recent advantages in evolutionary fuzzing has shown very promising results in security testing [10, 40, 42, 44, 51]. TIFF is an evolutionary fuzzer and as a result, there are existing fuzzers that come closer to TIFF in their functionalities.

In design closest to our proposal is VUzzer, which also uses DTA to infer important input properties for *smart* fuzzing. However, as mentioned earlier in this paper, TIFF's unique type-based mutation makes it much more powerful than VUzzer. In a very very recent work (S&P, May, 2018 [14]), Chen *et al.* proposed Angora- a fuzzer which uses taintflow analysis, but at the source code level by using LLVM's *DFSan* analysis tool, whereas TIFF works directly on the binaries of the applications.

AFLFast [10], which improves AFL's input generation strategy, applies a probabilistic approach to prune *uninteresting* inputs, thereby speeding up the generation of *interesting* inputs. Similarly, a very recent work by Gan *et al.* (CollAFL) improves AFL by considering the *path connectivity* of the executed path, i.e., selecting an input that corresponds to a path that has more uncovered *neighboring* branches. In contrast, TIFF learns which offsets are *intereting* to fuzz and what type of mutation should be applied to achieve better coverage. Our experimental results shows that TIFF outperforms AFLFast on every application that we tested.

On a different spectrum, there have been approaches that apply symbolic execution for input generation [12, 26, 47]. Driller [47], for example, uses AFL together with a concolic execution engine (based on angr [43]) to drive the input generation. The combination of evolutionary fuzzing and symbex has shown good results on DARPA CGC [17]. TIFF substantially differs from such approaches as its input generation depends on DTA and its mutation strategy is also tuned to certain class of bugs.

9 CONCLUSIONS

In this work, we elaborate on challenges faced by current fuzzers while mutating the input. The main challenge comes from the fact that fuzzers unaware of the type of offsets in the input resort to inefficient random mutation. This work argues that this mutation component is crucial and responsible both for triggering bugs and increasing code coverage. Therefore, we show that by inferring types, and associating them with every offset of the input, we can prioritize important *offsets* as well as *values* at those offsets to improve code coverage, but also to increase the probability of triggering bugs.

Specifically, we proposed a new mutation strategy that uses input type inference for achieving excellent code coverage, while trying to also maximize the coverage of bugs. We implemented the proposed mutation strategy in an effective, fully automated, input type-assisted fuzzer called TIFF, and evaluated our prototype on several real-world applications as well as the LAVA dataset. We compared the performance of TIFF with two state-of-the-art fuzzers, VUzzer and AFLFast, and showed that TIFF performs better than either of them with an order of magnitude fewer inputs. The concrete lesson we learn from our evaluation is that inferring input types by analyzing application behavior is a viable and scalable strategy to improve fuzzing performance.

ACKNOWLEDGEMENTS

We would like to thank the anonymous reviewers for their comments. This project was supported by the European Union's Horizon 2020 research and innovation programme under grant agreement No. 786669 (ReAct) and by the Netherlands Organisation for Scientific Research through grants NWO 639.023.309 VICI "Dowsing" and NWO 639.021.753 VENI "PantaRhei". This paper reflects only the authors' view. The funding agencies are not responsible for any use that may be made of the information it contains.

REFERENCES

[1] 2013. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-7437.
[2] 2013. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-1953.
[3] 2015. https://bugs.debian.org/cgi-bin/bugreport.cgi?bug=779849.
[4] 2016. https://gitlab.com/esr/gif2png/issues/1.
[5] 2016. https://bugs.freedesktop.org/show_bug.cgi?id=85141.
[6] 2016. https://bugs.debian.org/cgi-bin/bugreport.cgi?bug=844626.
[7] 2017. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-9167.
[8] Joy Arulraj, Guoliang Jin, and Shan Lu. 2014. Leveraging the Short-term Memory of Hardware to Diagnose Production-run Software Failures. In *Proc. ASPLOS '14*. ACM, 207–222.
[9] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed Greybox Fuzzing. In *Proc. CCS'17*. ACM, 2329–2344.
[10] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2016. Coverage-based Greybox Fuzzing As Markov Chain. In *CCS'16*. ACM, 1032–1043.
[11] Juan Caballero, Heng Yin, Zhenkai Liang, and Dawn Song. 2007. Polyglot: Automatic Extraction of Protocol Message Format Using Dynamic Binary Analysis. In *Proc. CCS '07*. ACM, 317–329.
[12] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. 2006. EXE: Automatically Generating Inputs of Death. In *CCS'06*. ACM, 322–335.
[13] S. K. Cha, M. Woo, and D. Brumley. 2015. Program-Adaptive Mutational Fuzzing. In *S&P'15*. 725–741.
[14] Peng Chen and Hao Chen. 2018. Angora: efficient fuzzing by principled search. In *IEEE S&P'18*. San Francisco, CA, USA.
[15] Lucian Cojocar, Taddeus Kroes, and Herbert Bos. 2017. JTR: A Binary Solution for Switch-Case Recovery. In *In Proc. ESSoS'17*. 177–195.
[16] Weidong Cui, Helen J. Wang, Marcus Peinado, Luiz Irun-briz, and Karl Chen. 2008. Tupni: Automatic Reverse Engineering of Input Formats. In *Proc. CCS'08*.
[17] DARPA CGC. 2015. DARPA Cyber Grand Challenge Binaries. https://github.com/CyberGrandChallenge.
[18] 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 *IEEE S&P'16*.
[19] Kathleen Fisher, Yitzhak Mandelbaum, and David Walker. 2006. The Next 700 Data Description Languages. In *POPL'06*. 2–15.
[20] Jonathan Foote. 2013. CERT Triage Tools.
[21] S. Gan, C. Zhang, X. Qin, X. Tu, K. Li, Z. Pei, and Z. Chen. 2018. CollAFL: Path Sensitive Fuzzing. In *IEEE S&P'18*. IEEE, 660–677.
[22] Vijay Ganesh, Tim Leek, and Martin Rinard. 2009. Taint-based Directed Whitebox Fuzzing. In *ICSE'09*. 474–484.
[23] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: directed automated random testing. *SIGPLAN Not.* 40, 6 (2005), 213–223.
[24] Patrice Godefroid, Michael Y. Levin, and David Molnar. 2008. Automated Whitebox Fuzz Testing. In *NDSS'08*. Internet Society.
[25] Patrice Godefroid, Hila Peleg, and Rishabh Singh. 2017. Learn&Fuzz: Machine Learning for Input Fuzzing. *CoRR* abs/1701.07232 (2017).
[26] Istvan Haller, Asia Slowinska, Matthias Neugschwandtner, and Herbert Bos. 2013. Dowsing for Overflows: A Guided Fuzzer to Find Buffer Boundary Violations. In *USENIX SEC'13*. 49–64.
[27] Christian Holler, Kim Herzig, and Andreas Zeller. 2012. Fuzzing with Code Fragments. In *USENIX Security 12*. Bellevue, WA, 445–458.
[28] Vasileios P. Kemerlis, Georgios Portokalidis, Kangkook Jee, and Angelos D. Keromytis. 2012. Libdft: Practical Dynamic Data Flow Tracking for Commodity Systems. In *SIGPLAN/SIGOPS VEE '12*. ACM, 121–132.
[29] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating Fuzz Testing. In *CCS'18*.
[30] Bogdan Korel. 1990. Automated software test data generation. *IEEE Transactions on software engineering* 16, 8 (1990), 870–879.
[31] libtiff. 2017. https://github.com/vadz/libtiff/blob/master/libtiff/tif_getimage.c#L267.
[32] Zhiqiang Lin, Xiangyu Zhang, and Dongyan Xu. 2010. Automatic Reverse Engineering of Data Structures from Binary Execution.. In *NDSS'10*.
[33] Zhiqiang Lin, Xiangyu Zhang, and Dongyan Xu. 2010. Reverse Engineering Input Syntactic Structure from Program Execution and Its Applications. *IEEE Trans. Softw. Eng.* 36, 5 (Sept. 2010), 688–703.
[34] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *PLDI'05*. ACM, 190–200.
[35] 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.
[36] David Molnar, Xue Cong Li, and David A. Wagner. 2009. Dynamic Test Generation to Find Integer Bugs in x86 Binary Linux Programs. In *USENIX Sec'09*. 67–82.
[37] Marcello Pagano and Kimberlee Gauvreau. 2000. *Principles of biostatistics* (2nd ed ed.). Australia ; Pacific Grove, CA : Duxbury.
[38] G. Ramalingam, John Field, and Frank Tip. 1999. Aggregate Structure Identification and Its Application to Program Analysis. In *Proc. POPL '99*. ACM, 119–132.
[39] Sanjay Rawat. 2016. VUzzer—Open Source Release. https://github.com/vusec/vuzzer.
[40] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. 2017. VUzzer: Application-aware Evolutionary Fuzzing. In *NDSS*.
[41] rc0r. 2017. Utilities for automated crash sample processing with AFL. https://github.com/rc0r/afl-utils/blob/master/afl_utils/afl_collect.py.
[42] Kostya Serebryany. [n. d.]. LibFuzzer: A library for coverage-guided fuzz testing (within LLVM). At: http://llvm.org/docs/LibFuzzer.html.
[43] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2016. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *IEEE S&P'16*.
[44] Zisis Sialveras and Nikolaos Naziridis. 2015. Choronzon: An approach at knowledge-based evolutionary fuzzing. https://github.com/CENSUS/choronzon.
[45] Asia Slowinska, Traian Stancescu, and Herbert Bos. 2011. Howard: A Dynamic Excavator for Reverse Engineering Data Structures. In *NDSS'11*.
[46] Asia Slowinski, Traian Stancescu, and Herbert Bos. 2012. Body Armor for Binaries: Preventing Buffer Overflows Without Recompilation. In *USENIX ATC'12*. USENIX, 125–137.
[47] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2016. Driller: Augmenting Fuzzing Through Selective Symbolic Execution. In *NDSS'16*. Internet Society, 1–16.
[48] Spandan Veggalam, Sanjay Rawat, Istvan Haller, and Herbert Bos. 2016. IFuzzer: An Evolutionary Interpreter Fuzzer using Genetic Programming. In *ESORICS*.
[49] Junjie WANG, Bihuan CHEN, Lei WEI, and Yang LIU. 2017. Skyfire: Data-Driven Seed Generation for Fuzzing. In *IEEE S&P'17*.
[50] Tielei Wang, Tao Wei, Guofei Gu, and Wei Zou. 2010. TaintScope: A Checksum-Aware Directed Fuzzing Tool for Automatic Software Vulnerability Detection. In *IEEE S&P'10*.
[51] Michal Zalewski. 2014. American Fuzzy Lop. At: http://lcamtuf.coredump.cx/afl/.
[52] Michal Zalewski. 2016. Integer Mutation. https://github.com/mirrorer/afl/blob/master/config.h#L223.

APPENDIX

9.1 Mutation Cycle Algorithm

Algorithm 1 shows a step-by-step procedure to create newer inputs, based on the two different kinds of mutation, that is control- and data-offsets based mutations. The following macros are used in the algorithm. For a given input ***i***, the functions `compute_(Howard|Tupni|Reward)(i)` calculate type inference for offsets in the input ***i***. `FUZZ_RUN` specifies the terminating condition for the given fuzzing run. `GET_FITNESS(i)` calculates the fitness of the given input—VUzzer's fitness function in our current prototype. `DATA_MUT_FREQ` specifies the number of generations that are skipped before we use data-offset based mutation. In our current prototype, we set this value to 10 (empirically evaluated as the optimal one). `CONTROL_OFFSET_MUTATE(i, O)` mutates the given input ***i*** by only targeting offsets that are used in some `cmp` instruction, along with their types. Similarly, `DATA_OFFSET_MUTATE(i, O)` mutates any offsets, along with their corresponding types.

9.2 Howard Implementation Details

As mentioned in Section 6, a significant part of our taint based input type inference system is based on Howard. HOwever, to make it suitable for our purposes, we modified it in several ways. In the following, we provide such details.

- As Howard identifies data structures in memory, to track taint from the input we associate a data structure with each memory address/register that keeps

Input: SI- set of initial seed inputs
1 **for** $s \in SI$ **do**
2 $I_H \leftarrow compute\_Howard(s)$;
3 $I_T \leftarrow compute\_Tupni(s)$;
4 $I_R \leftarrow compute\_Reward(s)$;
5 **end**
Data: Let SP- set of inputs that executes basic blocks, not seen in earlier executions.
BI- set of inputs with best fitness score. Initialize $SP \leftarrow SI$
6 **while** *FUZZ_RUN* **do**
7 $IN = SP \cup BI$;
8 **while** $|D_t| < NUM\_PER\_GEN$ **do**
9 $O_D \leftarrow \phi$;
10 i =SELECT_RANDOM(IN);
11 $O_D = GET_OFFSETS(i)$;
12 $i\prime = CONTROL\_OFFSET\_MUTATE(i, O_D)$;
13 $D_t \leftarrow i\prime$;
14 **end**
15 FT- dictionary of input with their fitness score;
16 **for** $i \in D_t$ **do**
17 RUN(i);
18 **if** *i executes a new BB* **then**
19 $SP \leftarrow i$;
20 $I_T \leftarrow compute\_Tupni(i)$;
21 $I_R \leftarrow compute\_Reward(i)$;
22 **end**
23 $FT \leftarrow GET\_FITNESS(i)$;
24 **end**
25 $BI = TOP(FT)$;
26 **if** *GEN_NUM % DATA_MUT_FREQ == 0* **then**
27 **for** $s \in SP$ **do**
28 $O_D = GET_OFFSETS(s)$;
29 $DI = DATA\_OFFSET\_MUTATE(i, O_D)$;
30 **for** $d \in DI$ **do**
31 RUN(d);
32 **end**
33 **end**
34 **end**
35 **go to** 6;
36 **end**
37 **Def** *GET_OFFSETS (input)*
38 $O = \phi$;
39 **if** $input \in I_H$ **then**
40 $O = O \cup I_H[input]$;
41 **end**
42 **if** $input \in I_T$ **then**
43 $O = O \cup I_T[input]$;
44 **end**
45 **if** $input \in I_R$ **then**
46 $O = O \cup I_R[i]$;
47 **end**
48 return O;

Algorithm 1: Steps involved in control- and data-offsets based Mutation

track of the tainted tag. We use a compressed bitset data type[6] data structure to reduce the memory footprint with little performance overhead.

- Our DTA framework is based on libDFT [28] which originally worked only on 32 bit systems. To make it suitable for 64 bit systems, we extended its tagmap structure to support 64-bit systems. Libdft stores tainted data tags in a tagmap, which contains a process-wide data structure (shadow memory) for holding tags of data stored in memory, as well as a thread-specific structure to hold tags for data residing in CPU registers. In addition, while libDFT did not support `xmm` registers, we store tags for both general purpose and `xmm` registers in the *vcpu* architecture which is a part of the tagmap structure. The tagmap holds multiple vcpu structures, one for every thread of execution. For capturing taint at the byte level, we need 8 tags for every 64-bit general purpose register and 16 tags for every `xmm` register.
  As mentioned, each tag is a compressed bitset (*EWAHBoolArray type*) data structure which stores the `file-offset` that affects a particular byte of the structure under consideration. Libdft stores memory tags in dynamically allocated tagmap segments. These segments are allocated dynamically, as and when requested by the application when making system calls such as `mmap()`, `read()`. During initialization, libDFT allocates a segment translation table to map the virtual address to the tags present in these tagmap segments. Therefore, for some applications, tagmap segments may overflow if the memory usage is very high or if the file size is very large.
  During our experimentation, we observed that in the later stages of application execution tagmap segment overflows occur quite regularly, slowing down the analysis significantly. For this reason, we have implemented a configurable timeout on the taint propagation during application execution which is configurable. After some profiling, we set this timeout to $MAX\{2 \times (execution\text{-}time\text{-}on\text{-}seed\text{-}input), 10min\}$, which performs well in our experiments.
- To identify data offsets types in the tainted input, we also incorporated Tupni algorithm in Howard, as an offline analysis phase, as described in section 4.2.
- We manipulated the array detection in Howard to get the more precise array types (Section 4.3) by utilising the results from DTA .
- In our DTA, we added callbacks for various libc functions such as `strcpy`, `memcpy`, etc., to detect strings which are tainted by input offsets. Such offsets are good targets for buffer-overflow related mutations. Similarly, we monitored functions such as `strcmp, memcmp`, etc., to get strings which are used for comparison in the application. Doing so, allows TIFF to infer *interesting* offsets that can be used in mutation to execute different paths.

9.3 Crash Analysis Details

In this section, we provide more insight on our crash analysis results.

Table 5 presents the results of running `!Exploitable` on the crashes found by TIFF. Table 4 provides information on the type of bugs (as discovered by `!Exploitable`) that TIFF is able to trigger on fuzzed applications. We can observe in Table 4 that several of the reported crashes are due to the invocation of the `abort()` call. On further investigation, we found that these applications are protected by gcc's cookie-based hardening option. While not exploitable, TIFF empirically shows the presence of such bugs also.

Table 5: Percentage of exploitable bugs discovered by TIFF as reported by !Exploitable tool.

Program	Unknown	Exploitable	Probably Not Exploitable	Probably Exploitable
`mpg321`	0.00	100.00	0.00	0.00
`pdf2svg`	100.00	0.00	0.00	0.00
`jbig2dec`	10.71	75.00	10.71	3.57
`potrace`	30.76	61.53	0.00	7.69
`gif2png`	100.00	0.00	0.00	0.00
`tcptrace`	0.00	66.66	33.33	0.00
`autotrace`	54.54	36.36	9.09	0.00
`pdftocairo`	100.00	0.00	0.00	0.00
`convert`	100.00	0.00	0.00	0.00

We further analyzed the quality of the bugs discovered by TIFF, by manually running each crash-triggering input with GDB to analyze the crash. We observed that 3 of the crashes in potrace occurred inside libpotrace. In the case of jbig2dec, convert and autotrace, all the crashes happened inside libjbig2dec, libMagickCore and libautotrace

[6] https://github.com/lemire/EWAHBoolArray

respectively. Bugs in libraries are more serious than those in the application code itself, as the buggy libraries may be used by other applications too. For pdf2svg and pdftocairo, one crash occurred inside libcairo and the other crash in libpoppler. For mpg321, two crashes happened inside libid3tag and for gif2png all crashes occurred inside the main application.

9.4 Results on MA dataset for 24hr Run

In a recent paper by Klees *et al.* [29], the authors evaluated several fuzzing prototypes and as a results, made several recommendations for fuzzing experimentation. One of the recommendations is to run the fuzzer for the duration of 24hrs. As in our original experimentation, we ran TIFF for 12hrs, we report the performance of TIFF over a duration of 24hr run for each application. As can be noticed in the table 6, we do not see any significant difference between these two sets of experiments. We opine this behavior can be attributed to the *smart* mutation strategies adopted by TIFF. It should also be noted that this set of experimentation did not involve multiple runs for each application and we report the figures only for the single run.

Table 6: Performance of TIFF under the 24hrs run per application.

Application	#Unique crashes	#Inputs	#BBs
mpg321	3	37670	527
pdf2svg	2	24855	5575
jbig2dec	32	30343	1368
potrace	12	26452	1532
gif2png	13	30694	1374
tcptrace	4	50359	1552
autotrace	27	22142	1743
pdftocairo	3	26682	4830
convert(gif)	1	5859	5569