AIFORE: Smart Fuzzing Based on Automatic Input Format Reverse Engineering

Ji Shi[1,2,3,4], Zhun Wang[*2], Zhiyao Feng[2,6], Yang Lan[2], Shisong Qin[2], Wei You[5], Wei Zou[1,4], Mathias Payer[6], and Chao Zhang[†2]

[1]{CAS-KLONAT[‡], BKLONSPT[§]}, Institute of Information Engineering, Chinese Academy of Sciences
[2]Institute for Network Science and Cyberspace & BNRist, Tsinghua University; Zhongguancun Lab
[3]Singular Security Lab, Huawei Technologies
[4]School of Cyber Security, University of Chinese Academy of Sciences
[5]Renmin University of China
[6]EPFL

Abstract

Knowledge of a program's input format is essential for effective input generation in fuzzing. Automated input format reverse engineering represents an attractive but challenging approach to learning the format. In this paper, we address several challenges of automated input format reverse engineering, and present a smart fuzzing solution AIFORE which makes full use of the reversed format and benefits from it. The structures and semantics of input fields are determined by the basic blocks (BBs) that process them rather than the input specification. Therefore, we first utilize byte-level taint analysis to recognize the input bytes processed by each BB, then identify indivisible input fields that are always processed together with a *minimum cluster* algorithm, and learn their types with a *neural network* model that characterizes the behavior of BBs. Lastly, we design a new power scheduling algorithm based on the inferred format knowledge to guide smart fuzzing. We implement a prototype of AIFORE and evaluate both the accuracy of format inference and the performance of fuzzing against state-of-the-art (SOTA) format reversing solutions and fuzzers. AIFORE significantly outperforms SOTA baselines on the accuracy of field boundary and type recognition. With AIFORE, we uncovered 20 bugs in 15 programs that were missed by other fuzzers.

1 Introduction

Fuzzing is an efficient technique for finding vulnerabilities in programs. Typically, fuzzing produces a large number of inputs and feeds them to programs under test, detecting security violations. A smart and effective fuzzer relies on a high quality of inputs, which enables efficient exploration of the program state space and increases the chance of triggering vulnerabilities.

Knowledge of the input format is essential to generating high-quality inputs. The input format describes how the program expects the input bytes to be organized. Ideally, well-formatted inputs will be parsed and processed properly, achieving the desired results. Ill-formed inputs will be filtered out by sanity checkers in the program and discarded early. Therefore, the fuzzer can generate inputs following the format specification, which will help bypass some sanity checks in shallow code, and finally, reach and test deeper and more complex code. Besides, because the sanity mechanisms in programs are not always sound and complete, it is highly possible that some inputs against the format *specification* are not filtered out but passed to deeper code [1]. These inputs usually have a greater chance of triggering unexpected behavior or bugs, and therefore should be favored by the fuzzer. Thus, the fuzzer can produce valuable inputs for both exploration and exploitation purposes with the guidance of format knowledge.

There are abundant works on input format inference and format-guided smart fuzzing. In general, such solutions try to answer three core questions regarding the input format: (1) where the boundaries of different input fields are, (2) which types these fields belong to, and (3) how to utilize the knowledge of input format to guide fuzzing. However, existing solutions so far only address parts of these problems or have limitations.

Regarding the *input field boundary recognition*, existing studies [2–4] mainly rely on statistical analysis or dynamic taint analysis to group bytes processed by the same instruction into a unique field. Such solutions have several limitations. First, an instruction may process multiple input fields, e.g., in a loop, and thus these fields will be erroneously merged into one field. Further, a long field would be processed by different instructions and be erroneously split into multiple fields, because one instruction, in general, cannot process data beyond the machine word size (e.g., 4 bytes for a 32bit CPU). Lastly, identifying fields through statistical analysis requires lots of different inputs, which are often not available.

Regarding the *input field type identification*, existing solutions [5–7] in general rely on prior knowledge (e.g., parameter

[*]Zhun Wang is the co-first author.
[†]Chao Zhang is the corresponding author.
[‡]Key Laboratory of Network Assessment Technology, CAS
[§]Beijing Key Laboratory of Network Security and Protection Technology

types of some standard library calls like `strcpy`) to extract the type of fields being passed in. Nevertheless, such prior knowledge is scattered and requires intensive engineering efforts to convert to heuristic rules and apply them, which will also inevitably introduce false positives and false negatives. Moreover, such solutions can generally only recognize program variable types (e.g., `int`, `array`, and `string`) rather than semantic types (e.g., `magic number`, `size`, or `checksum`), since they do not model how the input fields affect the program's behaviors from the semantic level.

Regarding the *utilization of input format*, existing fuzzing solutions [2,8–10] in general use the input format to guide test case generation or mutation to reach deeper code in the target program. Besides, some solutions also use format knowledge to determine if a seed is good or not. For example, AFLSmart [10] assigns more power (i.e., performing more mutations) to well-formed seeds, which can be parsed successfully by Peach [11] (a generation-based fuzzer with manually crafted input specification templates). However, it cannot distinguish the quality of seeds if given a simple template without field relation constraints (Table 1).

Our approach: To address these challenges, we propose AIFORE which automatically reverses input formats, later using them to smartly guide fuzzing. The core insight is that, since input fields are interpreted by BBs, the *structural* and *semantic* information of inputs can be inferred from them, no matter what the input format specification is like. Thus, we can learn the structures and semantics of input fields by analyzing BBs that process them, and then utilize the knowledge to conduct smart fuzzing.

First, AIFORE utilizes dynamic taint analysis to learn which input bytes are processed by each BB. Note that, a single indivisible field may be partially processed by multiple instructions in one BB, but is unlikely to get partially processed by multiple BBs, since these BBs will not always get executed together but an indivisible field should get analyzed together. As a result, we assume that BBs can be used as a basis to identify field boundaries. Given the taint analysis results, we split field boundaries by recognizing indivisible fields via a *minimum cluster* algorithm. (§3.1)

Second, AIFORE builds a deep learning model to comprehend BBs, i.e., to predicate the type of input fields processed by them. Note that, BBs process different types of input fields (e.g., `size`, `offset`, `enum`, `string`, `checksum`, or `magic number`) differently. Therefore, we train a Convolutional Neural Network (CNN) model [12] to learn the patterns of how BBs process different types of input fields and then predict the semantic type of input fields. (§3.2)

Third, we design a novel power scheduling algorithm based on the format dynamically extracted during fuzzing to improve fuzzing efficiency. Note that, a program may still accept inputs that do not satisfy the specification [1], as well as inputs of different format variants. Different variants of input formats have different impacts on program behaviors. Therefore, we present a two-step strategy to utilize the format knowledge, i.e., recognizing new variants and prioritizing infrequently tested formats. First, we pick test cases that have significantly different code coverage and re-analyze their input formats, since they are likely to have different formats. Second, we prioritize the seeds whose formats are less frequently tested during fuzzing, and increase their mutation power. (§3.3)

Results: We compare AIFORE prototype with SOTA format-aware fuzzing solutions and input reversing works, including ProFuzzer [8], TIFF-fuzzer[1] [6], WEIZZ [9], EcoFuzz [13], AFL-Analyze [14], and AFLFast [15]. We conduct experiments on 15 different types of formats and 15 well-tested real-world programs, including document readers, multimedia processors, packet parsers, compression tools, and network protocol analyzers. The results show that AIFORE's format reverse engineering module achieves high accuracy for different seed sizes and different compiler optimization levels (§5.1). AIFORE's average accuracy of field boundary recognition is **84.06%**,[2] compared to the corrected ground truth from 010 Editor[3], while the accuracy of ProFuzzer, TIFF-fuzzer, and AFL-Analyze are 36.27%, 63.14%, and 23.73%, respectively (§5.2). Regarding the field type identification, AIFORE correctly predicts the type with an accuracy of **84.26%** in untrained formats and programs, higher than ProFuzzer's 56.60% and AFL-Analyze's 36.76% (§5.2). At last, we apply the extracted format knowledge and the power scheduling algorithm to fuzzing. In the testing of 24 hours, AIFORE outperforms other fuzzers in BB coverage (6% higher than ProFuzzer, 26% higher than WEIZZ) and finds 20 bugs missed by other fuzzers (§5.3).

In summary, we make the following contributions:

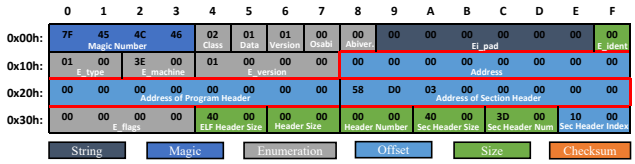
- We propose a field boundary recognition method, which utilizes taint analysis to identify relationships between input bytes and BBs, as well as a *minimum cluster* algorithm to split indivisible input fields.
- We present a novel deep learning–based solution to predict the type of input fields processed by BBs.
- We present a novel format-based power scheduling algorithm to explore infrequent types of inputs.
- We implement a novel smart fuzzing solution AIFORE and systematically evaluated it on a wide range of input formats and programs. Results show that it has a much better performance on input format recognition and format-aware fuzzing than SOTA solutions.

2 Motivational Example

In this section, we take `readelf` as an example to illustrate how it parses the `ELF` file format and summarize the limitations of existing approaches.

[1] We use TIFF-fuzzer to distinguish it from the file format `TIFF`.
[2] We calculate the arithmetic mean of the accuracy in Table 6.
[3] A popular commercial editor which can resolve different file formats. Homepage: https://www.sweetscape.com/010editor/

(a) ELF file header structure of 32bit.



(b) ELF file header structure of 64bit.

Figure 1: ELF file header definition.

An ELF file consists of several data structures (e.g., file header, program header, and section header), each composed of several fields. For the file header structure as shown in Figure 1, it has some fields consisting of consecutive bytes (e.g., magic number from offset 0x00 to 0x03) and some fields with single-byte values (e.g., class at offset 0x04). Take the e_machine at offset 0x12 as an example, it indicates the machine architecture (e.g., AArch64, i386, or x86_64), which changes the structure of ELF (e.g., the length of address fields can be four or eight bytes, marked with the red box).

Listing 1 shows the code snippet of readelf, which parses inputs of ELF format. We can see that there are two main steps to parse the input file. The first step is to read the input and initialize certain variables corresponding to the input fields. For example, line 3 to line 6 read the fields from the input and initialize corresponding variables (e.g., file_header.e_machine). The next step is to further process these initialized variables in the function process_file_header. Different BBs are used to process different types of variables (i.e., input fields). For example, the four-byte magic number is compared one byte at a time on line 21, while the e_machine is handled as an enumeration with a switch-case statement.

From this example, we observe the following:

Observation 1: In most cases, bytes in an indivisible field are parsed together in one BB. For example, the two-byte field e_machine is processed as a whole in the block starting at line 4, and another block starting at line 10. For the field e_shstrndx, it is parsed together in BBs at line 27. However, there are corner cases in which parts of one field (e.g., magic number field parsed at line 21) can be parsed in different BBs. Developers' code style and compiler optimizations would affect whether to parse them as one entity or separately. We will illustrate this further in §5.1.1.

Besides, one BB may process multiple fields and get executed multiple times at runtime. For example, BBs in the BYTE_GET function are executed multiple times to read from the input buffer and extract different fields to assign different variables.

Listing 1: Source code taken from readelf, which is responsible for parsing ELF format input.

```
1  #define BYTE_GET(field) byte_get(field, sizeof (field))
2  static bfd_boolean get_file_header (Filedata * filedata
       ){
3    ...
4    filedata->file_header.e_machine = BYTE_GET(ehdr32.
         e_machine);
5    filedata->file_header.e_shnum   = BYTE_GET(ehdr32.
         e_shnum);
6    ...
7  }
8  void init_dwarf_regnames_by_elf_machine_code(unsigned
       int e_machine){
9    dwarf_regnames_lookup_func = NULL;
10   switch (e_machine){
11     case EM_386:
12       init_dwarf_regnames_i386 ();
13       break;
14     case EM_X86_64:
15       init_dwarf_regnames_x86_64 ();
16       break;
17       ...
18     }
19 }
20 static bfd_boolean process_file_header (Filedata *
       filedata){
21     if (header->e_ident[EI_MAG0] != ELFMAG0 ||...||
           header->e_ident[EI_MAG3] != ELFMAG3){
22       return error;
23     }
24   ...
25   init_dwarf_regnames_by_elf_machine_code(filedata->
         file_header.e_machine)
26   ...
27   if (header->e_shstrndx != SHN_UNDEF && header->
         e_shstrndx >= header->e_shnum){
28       printf("corrupt");
29   }
30   ...
31 }
```

Observation 2: The program code processing fields of different types shows different patterns. For example, an enumeration variable (e.g., e_machine) is very likely to be processed by a switch-case statement, and a size variable (e.g., e_shnum) may be processed by mathematical operations.

Observation 3: The structure of the input may differ, and the program will dispatch distinct code to parse the input. For example, the variation of e_machine may indicate a different data structure, like the length of address fields marked with the red box in Figure 1. If a new structure is found during fuzzing, the fuzzer should re-assign the power and use the corresponding format knowledge to get more coverage.

From these observations, we summarize that the existing solutions have the following limitations. First, splitting fields at the instruction level is not generally appropriate. For example, instructions in BYTE_GET may parse multiple fields and will cause different fields to be merged erroneously. Second, most existing solutions of field type identification rely on human-extracted code patterns , which are labor-intensive and cannot cover complicated cases. For instance, there could be multiple BB patterns given one input field type. Third, existing format-aware fuzzing solutions have limitations when considering the power scheduling during fuzzing. For example, ProFuzzer [8]

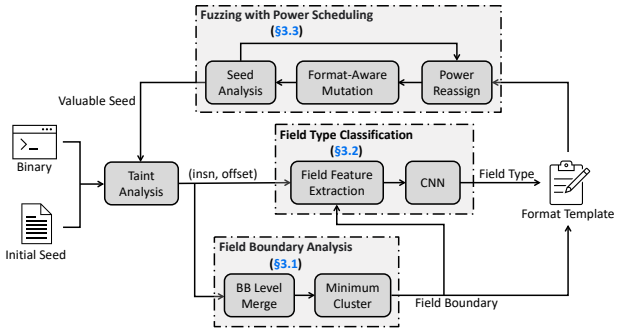


Figure 2: Architecture of AIFORE.

only analyzes the seeds it considers valuable to extract format knowledge and does not re-assign the fuzzing power to these valuable seeds. The validity-based power scheduling highly depends on Peach [11] and the constraints (e.g., checksum or size) in the `pit` file. Peach is a fuzzing framework that can generate program input with a given format model called a `pit` file. AFLSmart defines the degree of validity $v(s)$ of a seed s based on how many input bytes can be parsed by Peach and assigns more power to the seeds with higher $v(s)$. In the extreme case, if there is no constraint in the `pit` file, Peach will parse the input successfully in most cases and assign a 100% degree of validity to it. To verify this, we choose seven formats and target programs to observe to which degree the validity changes during fuzzing in AFLSmart. We choose these targets since AFLSmart provides the corresponding `pit` files, and they contain a few constraints, like checksum, size, and others.

We divide the experiments into two groups. For the first group, we launch AFLSmart with its original, provided `pit` files, containing the description of the constraints between fields. For the second group, we feed AFLSmart with customized `pit` files. These `pit` files only describe field boundaries and types, without complex constraints. We use AFLSmart to fuzz for 12 hours and collect thousands of seeds for both groups. We collect the number of seeds considered "valid" and assigned a high degree of validity by AFLSmart.

From the result (Table 1), we conclude that $v(s)$ is almost 100% when we feed AFLSmart with `pit` files that lack constraints. Almost all the seeds have the same degree of validity, which makes the power scheduling fall back to AFL, despite the format quality of the seeds varies. However, if we feed AFLSmart with well-prepared `pit` files, the distributions vary for different formats and programs. For `PNG`, the average of $v(s)$ decreases to 35.23%, and 29.90% of them are no less than 50%, which is the key threshold to judge the quality of a seed file in AFLSmart. In AIFORE, we design a new power scheduling algorithm to re-assign the energy to those formats that are seldom fuzzed.

Table 1: AFLSmart's degree of validity $v(s)$, average $v(s)$ / ratio of $(v(s) \geqslant 50\%)$

Format	Program[1]	`pit` w/ Constraints	`pit` w/o Constraints
ELF	readelf	0.15%/0.14%	100.00%/100.00%
GIF	gif2tiff	0.58%/0.59%	100.00%/100.00%
JPG	jhead	0.06%/0.06%	100.00%/100.00%
PCAP	tcpdump	3.48%/3.73%	100.00%/100.00%
PNG	pngtest	35.23%/29.90%	100.00%/100.00%
WAV	sfinfo	52.86%/54.27%	100.00%/100.00%
ZIP	7za	0.13%/0.13%	100.00%/100.00%

[1] Parameters are shown in Table 2

3 Design

The architecture of AIFORE is shown in Figure 2. During fuzzing, if the coverage increment a seed brings is distinguishable (i.e., increasing 3% of the average coverage), we then mark it as a valuable seed. For each valuable seed, we use AIFORE to analyze its format and fuzz the file with a format model including the extracted knowledge of field boundary and field type. The core insight of AIFORE is that the input format knowledge can be inferred from patterns of BBs processing the input file. Therefore, AIFORE utilizes taint analysis to construct a map between input bytes and the BBs processing them. Figure 3 demonstrates two BBs taken from `readelf` which parse the input format in the motivation example. The annotation at the end of each instruction lists the `offsets` of input bytes processed by it, traced by the taint analysis engine.

Given the knowledge of input bytes processed by each instruction, the following three modules jointly reverse the input format and balance the fuzzing power to the seeds. The first module analyzes the field boundary and splits the input into fields, i.e., consecutive bytes which have the same semantics concerning the impact on program behaviors. The second module predicts the field type information via a CNN-based model, which is trained to comprehend how a program parses different types of fields. The field boundary and field type consist of the format template (i.e., a peach pit file). For each field, we record its boundary with its start position and size and its type. The last module will decide which seed is worth format extraction and will re-assign more fuzzing power to those less mutated formats.

3.1 Field Boundary Analysis

Identifying the boundary of different fields is a fundamental task for reverse engineering of input formats. Given Observation 1, a field consisting of consecutive input bytes is usually processed as a whole in each BB. We thus propose to split fields with a minimum cluster (MC) method from the block level rather than the instruction level as Tupni [4] and Polyglot [3] did. The overall process is shown in Algorithm 1. First, we split the binary code into BBs. As shown in Figure 3, we will get two BBs from line 4 in Listing 1. Second, we collect and merge the input bytes processed in one BB. For instance, the first BB in Figure 3 processes two input bytes at offsets 18 and 19, and the second BB (`BYTE_GET` at line

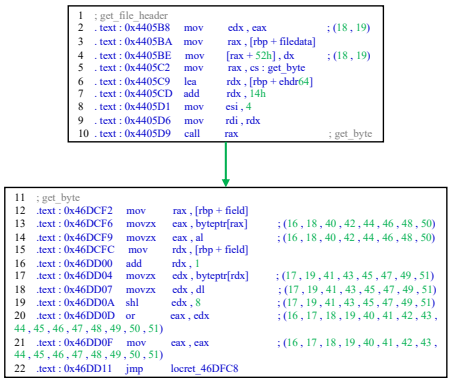


Figure 3: Basic blocks to parse `e_machine`.

Algorithm 1 Field Boundary Analysis

```
Input: BinaryProgram, TaintTrace
Output: FieldBoundaries                ▷ A list of field intervals without overlapping
BBs = ExtractBB(BinaryProgram)
BB_to_taint = Dict[BB, Set]            ▷ Map a BB to a set of tainted offsets
foreach inst ∈ TaintTrace do           ▷ Merge inst-level taint to BB-level
    BB = AddrToBB(inst.addr, BBs)
    taint_offset = ExtractTaintOffsetFromTrace(inst)
    BB_to_taint[BB].udpate(taint_offset)
FieldBoundaries = []
foreach taint_offset ∈ BB_to_taint.values() do
    interval_lst = SplitToConsecutive(taint_offset)
    foreach interval ∈ interval_lst do
        FieldBoundaries.append(interval)
        SplitOverlappedIntervals(FieldBoundaries)
                                       ▷ Split overlapped intervals with the sorted end-points
return Sorted(FieldBoundaries)
```

1 in Listing 1) processes bytes at offsets 16–19 and 40–51. The second BB is common in programs since they would first read different fields in the input as a whole to a buffer in the memory, then the program will parse them accordingly. Third, we split fields based on each block's taint attributes both in general BBs (e.g., `BYTE_GET`) and field-specific ones. For each MC of consecutive bytes, which is processed as a whole in all BBs, e.g., the bytes at (18, 19) in Figure 3, will be recognized as a field.

The MC method can find most input fields correctly. However, there are some exceptions that the field boundaries are different from the specification. This is because the program parses the field in its way. For example, the program may check the magic number byte by byte rather than as a whole. In such cases, MC may fail to group the bytes into a single field. We will explain it with an actual example in §5.1.1.

3.2 Field Type Classification

Based on Observation 2, we propose to classify the pattern of BBs to infer the type of input fields processed by them. Previous works [16, 17] have proved that neural network models are effective at extracting hidden features and outperforming humans in several classification tasks. Thus, here we utilize neural networks to model BBs that process input fields.

Specifically, given a field type and the code snippets that parse and use the fields of this type, we build a CNN model to map the latter to the former. We, therefore, collect a large number of training data consisting of code snippets and the types of fields processed by them. Then we train a CNN model to predicate the field type from a target code snippet.

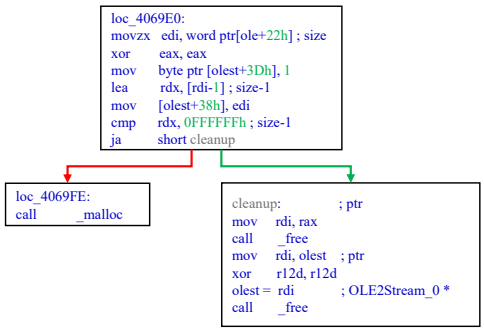


Figure 4: Forward slicing example.

3.2.1 Field Types

We consider the semantic types (e.g., `offset` or `checksum`) of fields rather than their program variable type [18, 19] (e.g., `int` or `string`). Such semantic types are more challenging to identify than variable types but are more beneficial to fuzzing. In our prototype, we support six semantic types which can help the fuzzer mutate the field more efficiently for most real-world programs.

- **Size.** This type of field represents the length of a data chunk consisting of one or multiple fields in the input file. For example, the `e_ehsize` field at offset 0x28 in Figure 1 indicates the header size is 0x34 bytes.
- **Enumeration.** This type of field can only take a limited set of valid values. For example, the `e_type` field indicates the binary type, taking one of the seven valid values defined in the `ELF` format [20].
- **Magic number**. This type of field, in general, serves as the signature of a file. For example, the first four bytes of an `ELF` file is a magic number with the value "\x7fELF".
- **String.** This type of field indicates a string literal, either in ASCII, Unicode, or other encodings.
- **Checksum.** This type of field is used to verify the integrity of a part of data in the input and is common in media files, compressed files, and font formats (e.g., `PNG`, `ZIP`, or `TTF`).
- **Offset.** This type of field indicates the location of another chunk of data in the input. For example, the `e_shoff` field at offset 0x20 in Figure 1 indicates the offset of the section header.

3.2.2 Training Data Collection

We use the input format knowledge provided by the 010 Editor tool as the ground truth. We pick some well-known formats and programs that process them, then perform the taint anal-

Algorithm 2 Data Vectorization

```
Input: FieldOffset, BinaryProgram, TaintTrace
Output: FeatureOffset
1: IRFEATURE = ['CmpNEZ','MSubF', ... ]
2: LIBCALL = ['memcpy','strcpy', ... ]
3: FORMATSTR = ['%s','%d', ... ]
4: BBs = ExtractBB(BinaryProgram)
5: FeatureOffset = [0]*(len(IRFEATURE)+len(LIBCALL)+len(FORMATSTR))
6: foreach BB ∈ BBs do
7:    if isBBProcessed(BB) then   ▷ Skip if BB has been processed because of forward slicing
8:       continue
9:    if isBBTainted(FieldOffset,TaintTrace,BB) then
10:      BB = ForwardSlicing(BB)
11:      ir_lst = ExtractIR(BB)   ▷ Vectorize IR features
12:      ir_feature = [0]*len(IRFEATURE)
13:      foreach operation ∈ ir_lst do
14:         index = IRFEATURE.index(operation)
15:         ir_feature[index] += 1
16:      str_lst = ExtractFS(BB)   ▷ Vectorize format string features
17:      str_feature = [0]*len(FORMATSTR)
18:      foreach str ∈ str_lst do
19:         index = FORMATSTR.index(str)
20:         str_feature[index] += 1
21:      libcall_lst = ExtractCALL(BB)   ▷ Vectorize libcall features
22:      call_feature = [0]*len(LIBCALL)
23:      foreach libcall ∈ libcall_lst do
24:         index = LIBCALL.index(libcall)
25:         call_feature[index] += 1
26:      FeatureOffset += (ir_feature,call_feature,str_feature)
27: return Normalize(FeatureOffset)
```

ysis. Given an input field, we determine its type from the ground truth, locate its offset, and then filter relevant BBs that have processed this field from the taint analysis results. For each relevant BB, we perform a forward slicing (i.e., line 10 in Algorithm 2) to merge consecutive child BBs with an important semantic dependency on the given field.

Take the code snippet in Figure 4 as an example. The `size` field is loaded in the first block and is compared against a threshold, then used by a `malloc` call in the consecutive child block. We tend to slice the code forward to merge these two blocks so that we can learn this field will be used by `malloc` and collect more meaningful features.

3.2.3 Data Vectorization

Algorithm 2 shows how AIFORE vectorizes the training data consisting of pairs of code slices. Several features are considered when vectorizing the code slices.

First, the semantic information of instructions in the BB is essential for determining how the program processes the field. To simplify the analysis, we transform instructions in BBs to Intermediate Representation (IR, e.g., VEX [21]). We vectorize the corresponding IR operation with one-hot encoding for each instruction that processes the given input field. Since the number of IR operations in VEX is too large for one-hot encoding, we select about 100 commonly used IR operations (line 1 in Algorithm 2).

Besides, we also consider the use of standard library calls as vital semantic information. We first choose a list of common standard library functions such as `memcpy`, `strcpy`, and `malloc`. For each invocation of such a library function, we will record it in the one-hot encoding.

Lastly, we record the format strings used in the BB and count them into the feature of the semantic information. For example, `%x` indicates an integer, while `%s` may imply a field of a string. This feature is also presented with one-hot encoding.

These three parts of the features above are concatenated (right side of line 26 in Algorithm 2) to get the feature vector of the slice. If a field is processed by multiple code slices, then these slices' vectors will be added together (left side of line 26) to get the overall feature vector of this field. As a result, we use the `(feature vector, field type)` pairs to train the neural network models.

Algorithm 3 Power Scheduling Algorithm

```
Input: Seeds
1: AvgFormatExecCnt = 0
2: foreach seed ∈ Seeds do
3:    if isInitialSeed() then
4:       seed.format = ExtractFormat(seed)
5:       seed.format.exec_cnt = 0
6:    if not FuzzerIsStuck() then
7:       MutateAndCheck(seed)
8:    else
9:       if seed.format.exec_cnt > AvgFormatExecCnt then
10:         continue   ▷ Skip the seed whose format has been mutated adequately
11:      else
12:         MutateAndCheck(seed)   ▷ Re-assign power to the seed with a less mutated format
13: procedure MutateAndCheck(seed):
14:    new_seed = MutateWithFormat(seed)
15:    ExecuteAndCheck(new_seed)   ▷ Execute 'new_seed' and add to queue if it brings new coverage
16:    new_seed.format = seed.format
17:    new_seed.format.exec_cnt = UpdateExecCnt()
18:    AvgFormatExecCnt = UpdateAvgCnt()
19:    if isValuable(new_seed) then
20:       new_seed.format = ExtractFormat(new_seed)
21:       new_seed.format.exec_cnt = 0
```

3.2.4 CNN Model Building

We choose the CNN model to infer the field type for two reasons. First, CNN models are proven effective in many classification tasks and binary analysis tasks [17, 22, 23]. In our task, we intend to classify the field types into different categories based on the extracted code features. Second, CNN models can naturally filter out the background information and effectively capture valuable features. In our task, the background information can be those features in the common instructions that are activated by multiple types of fields. In our prototype, we build a CNN model with six embedded hidden layers. We use `adam` as the optimizer, and use `categorical_crossentropy` as the loss function.

3.3 Fuzzing with Power Scheduling

Through the previous steps, we can identify field boundaries and field types. However, we still have the following two problems when using the knowledge to help fuzz. First, on which seed should we perform format extraction (i.e., power scheduling for format analysis)? Second, how can we bal-

ance the fuzzing power for different format variants produced during mutation (i.e., power scheduling for mutation)? We design a power scheduling algorithm as shown in Algorithm 3 , which is built on top of AFL, to solve both problems. First, AIFORE will try to analyze the initial seed to build its format model (Line 3 to 5). Then the fuzzer will mutate the seed with the guidance of format knowledge (Line 7). If the newly mutated seed is valuable, then we re-analyze its format (Line 19 to 21). When the fuzzer gets stuck (i.e., the fuzzer fails to get new coverage in a given time), we will assign the fuzzing power to those formats which are not fully mutated yet (Line 8 to 12).

Power scheduling for format analysis. The fuzzer generates numerous seeds during fuzzing. Identifying field boundaries and types of each seed is unnecessary and impractical, since not all the seeds are valuable, and analyzing all the seeds will cost too much power . Several works [13, 15, 24] have proved that not all seeds are valuable during fuzzing. So we extract the format knowledge (i.e., field boundary and field type) only for the initial seed and the seeds which we consider valuable. "Valuable" seeds in our design mean the input can reach more new BBs and likely belong to an unseen format variant (e.g., a new type of elf architecture in Figure 1). If a seed is "valuable", we extract its field boundaries and field types through the function `ExtractFormat` in Algorithm 3.

Power scheduling for mutation. Not all the formats are mutated equally during fuzzing. The fuzzer mutates the seed one by one with `MutateWithFormat` function. Specifically, it will mutate the bytes of each field as a whole , and choose a proper mutator based on the field type. For example, the mutator will try interesting size values instead of `bitflip` for a `size` type field and insert or delete bytes for a `string` type field. Besides, for those bytes without format knowledge, the fuzzer will use the default mutation methods like those in AFL. We will further analyze how the adaptive mutator benefits fuzzing in §5.3.2.

During mutation, whenever a valuable seed occurs, the fuzzer starts mutating the seed based on the re-analyzed format. However, if two "valuable" seeds show up close , the previous one is mutated inadequately. We design a power re-assign mechanism to balance the fuzzing power among the format variants. When the fuzzer gets stuck, we re-assign the fuzzing power to those seeds bound with less mutated format variants and skip those seeds bound with the fully mutated format variants.

4 Implementation

We build the taint analysis engine in C++ based on VUzzer [25] and libdft [26] to support 64-bit programs and byte-level taint analysis. We re-write the taint propagation component for each kind of instruction with about 5k lines of code to check if the operands are tainted by the input bytes and get the map of instructions and the input bytes processed by them. We develop two format analysis modules with 7K lines of Python code. We use Angr [27] as the backend to support static analysis. To train the model, we implement the machine learning component with Keras 2.2.4. To collect the ground truth data, we write an automation script to export the template result from 010 Editor with the help of AutoIt [28]. We convert the extracted format knowledge into a Peach `pit` file.

Table 2: (format, program) pairs used in our experiments.

Program&Parameter	Format	Boundary	Type[1]	Fuzz
7za t @@	7Z	✓	⊙	
	ZIP	✓	X*	✓
readelf -a @@	ELF	✓	⊙	✓
elfutils-readelf -a @@	ELF		X	✓
exiv2 pr @@	GIF		X*	✓
	JPG	✓	X*	
	TIFF		⊙	
tiffdump @@	TIFF	✓	⊙	✓
magick identify @@	GIF		⊙	✓
gif2tiff @@	GIF		⊙	✓
gifsicle @@ -o out.gif	GIF	✓	⊙	✓
jasper –input @@ –output test.bmp	BMP	✓	X	✓
jhead -v -exifmap @@	JPG		X	✓
pngtest @@	PNG	✓	X	✓
freetype_parser @@	OTF	✓	X*	
	TTF	✓	⊙	✓
sfinfo @@	WAV	✓	X	✓
tcpdump -nr @@	PCAP	✓	X	✓
xls2csv @@	XLS	✓	X	✓
arping 192.168.1.1 -I eno2 -c 1	ARP[2]	✓	X	
nslookup example.com	DNS[2]	✓	X	

[1] ⊙: the (format, program) pair is in the training set; X: the program is not in the training set; X*: the program is in the training set but the (format, program) pair is not.
[2] We show the evaluation results of `ARP` and `DNS` in §Appendix A.

5 Evaluation

We evaluate AIFORE to answer four research questions:

- **RQ1:** What is the performance of each format extraction module of AIFORE? (§5.1)
- **RQ2:** What is the format analysis performance of AIFORE, compared with other SOTA format reverse engineering works? (§5.2)
- **RQ3:** What is the performance of AIFORE in terms of code coverage and bug detection compared with other SOTA fuzzers? (§5.3)
- **RQ4:** How does each module of AIFORE contribute to fuzzing efficiency? (§5.4)

We run all experiments on a machine with a 24-core CPU (Intel(R) Xeon(R) CPU E5-2650) and 128GB memory. We train the model with a Tesla P100 GPU. After the model is trained, AIFORE only uses the GPU when predicting field types, and it is used infrequently.

5.1 RQ1: Performance of Format Extraction

In this section, we evaluate each of AIFORE's format extraction modules to see how well it extracts the format.

Target File Formats and Programs. According to Klees et al. [29], there are on average 7 real-world programs evaluated by 32 unique papers in recent years. In this paper, we collect 17 programs to evaluate AIFORE, as well as 15 kinds of formats, of which 13 are file inputs (including image, executable, compression file, or compound file) and 2 are network protocols (§Appendix A). Table 2 shows all formats and programs (with parameters) we used to evaluate each module of AIFORE.

We choose target inputs and programs based on the following considerations. First, these formats are diverse and widely used. Second, the program could parse as many fields as possible when given one type of input, since AIFORE analyzes input format based on the dynamic taint trace, and cannot recognize input fields that are not processed[4]. We compile these programs with different optimization levels (from `-O0` to `-Os`) to assess the robustness of AIFORE.

Ground Truth. First, given an input to test, we use the public format template from the 010 Editor to parse the file and export all field records with boundary and type information. These templates are written by experts and verified by the community.

Then, we manually preprocess the records to get the ground truth of field boundaries in two steps: (1) Delete redundant definitions from the ground truth. For example, Table 3 shows the signature field of the ELF format. Field records `file_identification[0]` to `[3]` should be removed since `file_identification[4]` exists as a whole field. (2) Remove field records that are not parsed by the target program. For example, `readelf` does not process the `ei_pad` field of the `ELF` format in Figure 1. We skip these fields because all approaches that reverse the input format based on the behavior of the program cannot figure them out.

Finally, we manually check the records to correct ambiguous semantic types of fields. For example, the `char cname[4]` field in the template for the `PNG` file indicates the type is `string`. However, it is used as a `magic number` type in the program `pngtest`. Another example, some formats may contain a `version` field, which could be confusingly marked as a `string` or an `integer`. To solve the problem, we manually analyze the semantic type of every field in the template, and remove those with confusing semantic type labels, increasing the accuracy of model training.

5.1.1 Field Boundary Accuracy

We denote the ground truth of boundary information as $Boundary_T$. We choose 5 samples from the Internet or from the test suite of the target programs which have various features like different file sizes, different compression levels, or different enumerations in specific formats. For each sample, we manually craft the ground truth of boundary information as described in **Ground Truth**. For each format, we choose one program which parses input files most completely as shown in column "Boundary" of Table 2. Then we use AIFORE to infer their field boundaries (denoted as $Boundary_A$).

Table 3: Signature field of `ELF` from 010 Editor.

Name	Value	Start	Size
char file_identification[4]	'\x7fELF'	0	4
char file_identification[0]	127'\x7f'	0	1
char file_identification[1]	69'E'	1	1
char file_identification[2]	76'L'	2	1
char file_identification[3]	70'F'	3	1

We use accuracy as the metric, i.e., the number of fields correctly identified by AIFORE divided by the total number of fields in the ground truth.

Note that $Boundary_T$ might be coarse-grained, i.e., the template from 010 Editor misses some fine-grained fields. For example, the template for `PCAP` only describes fields from the data link layer to the transport layer (i.e., TCP/UDP), without knowledge of fields in the application layer (e.g., HTTP/DNS). Nevertheless, the target program `tcpdump` parses the packet in more detail, making AIFORE produce more fine-grained boundary information. We skip these fields when calculating the accuracy. To ensure the fairness of the experiment, we also remove them when counting the accuracy for all other solutions in §5.2. Although AIFORE may perform worse (or even better) in the fields we manually removed, we consider the results will only be affected slightly since such cases are rare during our analysis. Further, our result in §5.2 shows the accuracy of AIFORE is much higher than other solutions.

The results are shown in Table 4 and we draw 2 conclusions. First, the accuracy is irrelevant to the compiler optimization level. That is because the MC method is considered from the view of semantic block, which is seldom affected by compiler optimizations. For the target of `7z` and `zip`, the total number of fields in the input is small, and thus even one incorrect field may cause a large accuracy fluctuation (about 10%). Second, we manually investigated the targets which are with low accuracy and found the reason is that the program parses the fields in its way. For example, `readelf` compares the first four bytes (i.e., `magic number`) in 4 BBs respectively, as shown in Figure 5. With the MC approach, we will split the magic number into four single-byte fields, as they are parsed in different BBs. However, they are defined as a whole in the specification. Although the result is different from the specification, we consider this will not affect (or even benefit) the fuzzing efficiency since fuzzing is to test the implementation of the program rather than extract accurate field boundaries compared to the specification.

[4]This limitation is shared among all solutions that utilize dynamic taint analysis [4, 8].

Table 4: Accuracy of field boundary analysis. The table shows the average number of fields and file size in $Boundary_T$, as well as the accuracy for different optimization levels of target programs.

Format	Program	Avg. Field Count	Avg. Size (bytes)	Optimation Levels				
				-O0	-O1	-O2	-O3	-Os
7Z	7za	9	425	55.56%	55.56%	55.56%	55.56%	55.56%
BMP	jasper	1,243	1,702	88.81%	89.38%	88.58%	89.14%	89.14%
ELF	readelf	77	324	96.10%	93.51%	93.51%	93.51%	93.51%
GIF	gifsicle	217	821	96.77%	96.77%	97.23%	97.23%	94.00%
JPG	exiv2	24	424	50.00%	50.00%	55.00%	53.58%	50.00%
OTF	freetype_parser	258	226,772	75.44%	76.64%	61.06%	61.06%	58.58%
PCAP	tcpdump	22	114	90.90%	90.90%	90.90%	88.64%	86.36%
PNG	pngtest	42	239	63.12%	64.31%	63.12%	63.12%	63.12%
TIFF	tiffdump	73	22,700	82.19%	82.19%	76.71%	76.71%	82.19%
TTF	freetype_parser	110	7,876	76.36%	76.36%	76.36%	76.36%	76.36%
WAV	sfinfo	18	37,524	77.77%	72.22%	72.22%	72.22%	72.22%
XLS	xls2csv	288	19,968	41.67%	41.67%	36.81%	36.81%	36.81%
ZIP	7za	12	906	58.33%	58.33%	58.33%	58.33%	58.33%
Average		**184**	**24,600**	**73.31%**	**72.91%**	**71.18%**	**70.94%**	**70.48%**

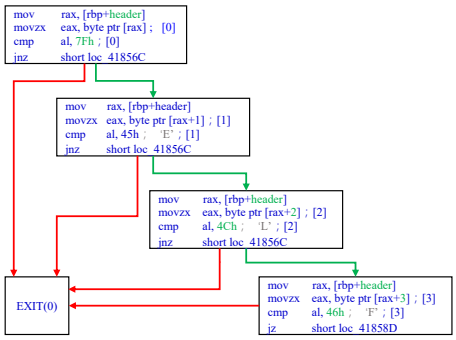


Figure 5: Basic block to parse `magic number`.

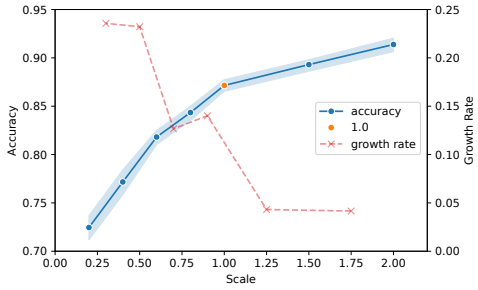


Figure 6: Mean validation accuracy (with 95% confidence interval error band) and growth rate among different scales of training set.

5.1.2 Field Type Accuracy

We now evaluate the performance of the CNN model based on 4 experiments.

• **Experiment 1: Can the CNN model perform well on the training set and the validation set?** We collect 10,582 fields from 8 formats, and we choose corresponding programs to parse these inputs to train the model. The training set is marked as ⊙ in column "Type" of Table 2. The ratio between the training set and the validation set we use is 4:1. Then we mark the six semantic types as labels and the corresponding (vectorized) taint traces as input data to train the CNN model. It takes, on average, 2-3 hours of manual effort to label the training data for one format. This only needs to be done once for each format and we consider this reasonable.

To validate whether the model can perform well under different compiler optimizations, we trained the model from `-O0` to `-Os` as training data respectively. To further investigate if the model applies to real-world cases (i.e., where the selected compiler optimization of the target is not known), we also evaluate the accuracy of the model with mixed optimizations. We spend 12 hours collecting the taint traces for all samples in the training set, and the training time for the models is on average 20 minutes. After training the model, we evaluate their Top-1 accuracy, i.e., whether the top-score prediction result is correct.

Figure 14 shows these results. We conclude that (1) the performance with different compiler optimization levels (i.e., `-O0` to `-Os`) behave differently, and the average accuracy is over 85% ; (2) the accuracy results are stable even with mixed compiler optimizations, which means the model applies to real-world cases.

• **Experiment 2: How much data do we need to train a sufficiently reliable CNN model?** To further analyze how much data is needed to train the CNN model, we also calculate the accuracy achieved by the model trained on different amounts of data. We set 10,582 fields as the unit (i.e., 1.0) scale and use different scales (i.e., 0.2 to 2.0) to train the model accordingly. For each group, we randomly choose the training data and repeat the experiment 5 times. The result is shown in Figure 6. From the result, we can conclude that a larger amount of training data results in higher model accuracy. However, the growth rate stagnates quickly when the scale is larger than 1.0.

• **Experiment 3: For trained programs, can the model predict field types of unseen formats?** We define a format is unseen by a program if the pair of (program, format) is not in the training set of AIFORE. In this experiment, we apply AIFORE to programs that have been trained with certain formats, and verify whether AIFORE could recognize unseen formats for these programs (marked as X* in Table 2). Specif-

Table 5: Field type classification accuracy. The results represent the Top-1 / Top-2 accuracy respectively, where the latter indicates whether the type of either the top two results is correct.

Program	Format	Samples	Optimization Levels					
			-O0	-O1	-O2	-O3	-Os	Mixed
exiv2	GIF	202	73.20%/80.00%	84.62%/100.00%	100.00%/100.00%	94.12%/100.00%	100.00%/100.00%	95.14%/100.00%
exiv2	JPG	109	77.78%/88.89%	90.00%/100.00%	66.67%/77.78%	80.00%/80.00%	72.73%/81.82%	76.66%/85.28%
freetype_parser	OTF	361	74.13%/86.42%	71.30%/85.19%	76.47%/87.50%	80.00%/90.00%	67.86%/89.29%	70.56%/92.23%
7za	ZIP	364	89.01%/96.98%	93.37%/97.79%	94.90%/99.06%	91.23%/98.25%	92.99%/97.19%	91.70%/98.18%
Average			**78.53%/88.07%**	**84.82%/95.75%**	**84.51%/91.09%**	**84.51%/92.06%**	**83.40%/92.07%**	**83.52%/93.92%**
elfutils-readelf	ELF	5385	62.66%/74.59%	73.93%/78.67%	61.11%/87.50%	54.99%/64.33%	71.69%/72.21%	67.33%/75.77%
jasper	BMP	1068	83.11%/97.30%	92.00%/99.43%	90.06%/99.42%	91.98%/100.00%	90.48%/98.10%	93.46%/100.00%
jhead	JPG	913	84.00%/88.00%	86.36%/86.36%	86.36%/90.91%	86.96%/86.96%	84.00%/92.00%	85.92%/86.92%
tcpdump	PCAP	1412	74.70%/91.42%	84.13%/85.72%	78.04%/91.77%	89.81%/91.40%	75.44%/81.43%	73.04%/85.83%
pngtest	PNG	1004	80.12%/87.77%	86.88%/91.10%	82.35%/85.88%	88.34%/89.11%	83.31%/88.96%	80.11%/82.31%
sfinfo	WAV	434	88.89%/100.00%	97.91%/100.00%	97.37%/100.00%	98.80%/98.80%	98.99%/100.00%	97.92%/99.74%
xls2csv	XLS	1339	71.00%/84.93%	75.36%/81.88%	68.00%/71.20%	71.94%/71.94%	65.56%/77.46%	66.95%/71.76%
Average			**77.78%/89.14%**	**85.22%/89.02%**	**80.47%/89.53%**	**83.26%/86.08%**	**81.35%/87.17%**	**80.68%/86.05%**

ically, for programs in the training set, we collected some new file formats supported by them, and apply our trained model to predict the field types of the new formats.

The upper part of Table 5 shows the evaluation result. We consider Top-2 accuracy because the Top-K suggestions from the model are still valuable for fuzzing. To be concrete, the Top-K field type knowledge can help the fuzzer reduce the mutation space and find high-quality test cases faster.

We have two findings from this table: (1) AIFORE can predict the field type in unseen formats with high accuracy, while the Top-1 accuracy is over 80% on average and the Top-2 accuracy is over 90%; (2) the model performance is irrelevant to the programs' optimization levels. Although compiler optimizations might change the features of the code, our model learns stable patterns across optimizations.

• **Experiment 4: For untrained programs, can the model predict their (unseen) input formats?** We apply the aforementioned trained model to analyze untrained programs (marked as `X` in Table 2) and predict their input formats. We choose 7 untrained programs (and 2 protocols shown in Appendix) and corresponding formats to test the accuracy of AIFORE. These untrained programs are chosen based on two criteria: (1) they should be able to parse the chosen formats, and (2) they do not share libraries, which process the chosen formats, with programs in the training set. The latter requirement is applied for a fair comparison. Note that, the chosen formats (e.g., `ELF`) may be processed by other trained programs (e.g., `readelf`) in the training set. But still, they are *unseen* to the untrained program (e.g., `elfutils-readelf`).

The bottom of Table 5 shows the evaluation result. We learn that AIFORE can predict the field type with a Top-1 accuracy of 81% and a Top-2 accuracy of 88% on average. Even when we use mixed test data from different compiler optimizations, AIFORE can also predict the field type based on how the program parses the field.

5.2 RQ2: Comparison of Format Extraction

In this section, we compare AIFORE with the CNN model trained from programs with mixed optimizations (i.e., `-O0` to `-Os`) against existing input format reverse engineering works, like Polyglot [3], Protocol Informatics (PI) project [30], ProFuzzer [8], AFL-Analyze [14], and TIFF-fuzzer [6] to measure the format reversing performance.

We collect the field boundary and the type results of AIFORE as described in §5.1.1 and §5.1.2, and compare the results with ProFuzzer, AFL-Analyze, and TIFF-fuzzer. There are also other tools like WEIZZ [9] that can extract input formats. However, we did not compare the field boundary accuracy of AIFORE with them since they cannot extract all the field boundaries even though the target program parses the fields already. We analyze concrete cases to explain the reason for such false negatives in §Appendix B.

Metrics. For the field boundary analysis, we calculate different solutions' accuracy as described in §5.1.1, i.e., the number of correctly identified fields divided by the total number of fields in the ground truth.

We delicately process the results for the field type analysis since different solutions focus on different field types. For example, ProFuzzer classifies the field types into 6 categories while AFL-Analyze recognizes only 3 semantic types. We manually check the result produced by different solutions and calculate the accuracy accordingly. For AFL-Analyze, we only check if its results match the 3 semantic types they define (i.e., `raw data`, `magic number`, and `length`). For ProFuzzer, we similarly check its results. For AIFORE, we manually check if the Top-1 result matches the 6 semantic types we define in §3.2.1.

In addition, we also measure the average time cost of analyzing one input format for each solution.

Test Targets. To be fair, we choose 4 formats and programs with different accuracy levels from the training set (Table 4) and from unseen formats and programs (Table 5) respectively. We intend to investigate what is the performance of other solutions regarding the targets with different accuracy levels under AIFORE. The chosen targets are shown in Table 6. For the programs, we compile all the targets with their default compiler optimizations. Regarding input samples, we choose the ones that are not in the training set for the trained programs and randomly choose some input samples for the untrained

Table 6: Field boundary/type accuracy comparison between different input format reverse engineering solutions.

Format		Program	Large Seeds					Small Seeds				
			Size(bytes)	AIFORE	ProFuzzer	AFL-Analyze	TIFF-fuzzer	Size(bytes)	AIFORE	ProFuzzer	AFL-Analyze	TIFF-fuzzer
Trained	ELF	readelf	808	96.43%/87.73%	37.40%66.25%	43.73%/40.00%	94.30%/(N/A)	324	98.91%/91.23%	74.26%/66.67%	55.70%/38.60%	97.40%/(N/A)
	GIF	gifsicle	695	97.64%/87.31%	12.59%/52.94%	6.44%/11.76%	71.96%/(N/A)	198	97.64%/72.23%	50.29%/60.00%	31.96%/12.00%	65.30%/(N/A)
	TIFF	tiffdump	448	82.23%/89.33%	27.13%/42.11%	13.19%/21.05%	81.30%/(N/A)	166	84.33%/88.45%	37.01%/40.00%	21.05%/21.67%	82.33%/(N/A)
	TTF	freetype	542	67.43%/83.22%	30.28%/65.69%	9.93%/20.44%	5.56%/(N/A)	148	72.23%/85.32%	2.20%/0.00%	1.35%/21.21%	40.00%/(N/A)
Untrained	PCAP	tcpdump	894	88.64%/82.34%	28.84%/71.14%	0.00%/39.04%	81.20%/(N/A)	114	93.18%/84.23%	73.18%/77.78%	39.66%/44.44%	85.60%/(N/A)
	WAV	sfinfo	572	100.00%/95.55%	63.85%/66.67%	48.77%/57.69%	100.00%/(N/A)	44	100.00%/91.22%	56.25%/63.64%	56.25%/45.45%	100.00%/(N/A)
	BMP	jasper	630	45.34%/83.54%	12.11%/91.18%	1.35%/32.35%	24.24%/(N/A)	58	45.34%/84.33%	75.00%/82.35%	28.12%/23.53%	22.22%/(N/A)
	XLS	xls2csv	6656	81.25%/74.56%	(N/A)/(N/A)*	22.16%/0.56%	25.50%/(N/A)	5632	94.40%/78.32%	(N/A)/(N/A)*	0.00%/51.02%	33.33%/(N/A)
Average			**1406**	**82.37%/85.45%**	**26.53%/57.00%**	**18.20%/27.86%**	**60.51%/(N/A)**	**836**	**85.75%/84.42%**	**46.02%/48.81%**	**29.26%/32.24%**	**65.77%/(N/A)**

* The tool fails to get the result within 24 hours

Table 7: Average time to parse a file (seconds).

Input		Program	Large Seeds					Small Seeds				
			Size(bytes)	AIFORE (B/T)	ProFuzzer	AFL-Analyze	TIFF-fuzzer[2]	Size(bytes)	AIFORE (B/T)	ProFuzzer	AFL-Analyze	TIFF-fuzzer[2]
Trained	ELF	readelf	808	29(24/5)	14,224	14	8	324	28(22/6)	977	6	7
	GIF	gifsicle	695	24(19/5)	84,123	42	91	198	23(17/6)	11,392	5	18
	TIFF	tiffdump	448	43(39/4)	1,529	9	13	166	44(40/4)	145	4	11
	TTF	freetype	542	19(12/7)	2,967	8	13	148	17(11/6)	100	2	6
Untrained	PCAP	tcpdump	894	24(19/5)	26,241	18	8	114	20(14/6)	95	2	8
	WAV	sfinfo	572	21(17/4)	2,993	11	8	44	23(19/4)	27	1	8
	BMP	jasper	630	21(15/6)	2,004	10	6	58	9(3/6)	110	0.1	6
	XLS	xls2csv	6,656	61(56/5)	(N/A)[1]	255	22	5,632	52(47/5)	(N/A)[1]	17	94

[1] The tool fails to get the result within 24 hours
[2] The time only includes field boundary extraction

programs. The details of the training set and validation set for field type prediction are described in the first part of §5.1.2.

Input Size. The input size will affect the run-time performance of format reverse engineering. AIFORE, ProFuzzer, and AFL-Analyze all rely on dynamic analysis to predict the field type, but with different methods. Both ProFuzzer and AFL-Analyze mutate each input byte and rerun the program to get the coverage bitmap as the profile of the current execution. Based on the variation of this profile, they can analyze the type features of each byte and combine consecutive bytes of similar features as a field with the corresponding type. Thus, a larger file may consume more time to get the result. AIFORE is based on the taint trace and the CNN model to infer the field boundaries and field types. Although the model is only trained once, the taint analysis requires a time-consuming analysis. To better understand how the input size affects the performance of each work, we split the input files into different groups according to their size and observe the time to complete the analysis in each group.

Results. Table 6 demonstrates the accuracy of field boundary and type analysis, and Table 7 shows the average cost of time to parse an input, respectively.

From Table 6, we can learn that AIFORE achieves higher accuracy both in field boundary recognition and field type prediction. Besides, AIFORE performs better in trained programs than those in untrained. This is consistent with our experience since the model has learned the pattern of how those programs parse different field types accurately.

The average time to parse a file with AIFORE (and TIFF-fuzzer) does not differ significantly for different size classes, while ProFuzzer and AFL-Analyze spend much more time parsing larger files. In Table 7, **B** represents boundary identification, the count includes the required time for taint analysis (which requires most of the total time). **T** represents type prediction, its time cost remains stable during the test since we have a trained model with a stable prediction time. We find that in Profuzzer and AFL-Analyze, the profiling phase consumes most of the time, which is strongly related to the input size. However, the taint analysis in AIFORE (and TIFF-fuzzer) is not very sensitive to the input size. Since AFL-Analyze and TIFF-fuzzer perform some rough analysis, they have better execution time but achieve lower accuracy than AIFORE on average.

Moreover, we also conduct reverse engineering on two protocols. The detailed results are presented in Appendix A. Compared with other tools [3, 30, 31], AIFORE can give not only more accurate format knowledge in terms of field boundary and type but also with more details.

5.3 RQ3: Comparison of Fuzzing Performance

There are already some fuzzers [2, 6, 8, 13, 15, 32] that try to extract format knowledge and perform power scheduling to optimize the fuzzing process. We compare AIFORE against them to investigate how much our format analysis and power scheduling can improve fuzzing efficiency. Note that for the field type classification, once the model has been built, we do not need to retrain the model during the fuzzing process.

Target Programs and Seeds. For the programs, we use 15 programs to parse files as shown in Table 2, of which 6 are trained and 9 have not been seen by AIFORE. For each file type, we randomly choose one input file as the initial seed for all fuzzers.

Fuzzers. We compare AIFORE with 6 fuzzers, including format-aware fuzzers, format-unware but popular fuzzers, and power scheduling optimization fuzzers, i,e, AFL [33], AFLFast [15], ProFuzzer [8], TIFF-fuzzer [6], WEIZZ [9], and EcoFuzz [13]. AFL is one of the most popular grey-box fuzzing tools, and there are many fuzzers built on top of AFL. AFLFast and EcoFuzz optimize AFL by prioritizing the seeds that may lead to new coverage. ProFuzzer has a dynamic

Table 8: Basic block coverage after 24 hours.

Format	Program[1]	AIFORE (B)[2]	AIFORE (B+T)[2]	AIFORE (B+T+P)[2]	AFL	AFLFast	EcoFuzz	ProFuzzer	TIFF-fuzzer	WEIZZ
BMP	jasper	7123	7705	**7887**	6694	5926	7777	7437	5331	6344
ELF	readelf*	9880	13798	**17985**	9652	12020	13791	17872	2426	15720
	elfutils-readelf	7213	7541	**8308**	6995	5873	6608	7978	2180	6519
GIF	gifsicle*	4702	5062	**5435**	4528	4634	4675	5315	4146	4578
	magick*	14221	14414	**16685**	13529	13684	12335	14512	8351	10002
	gif2tiff*	2458	2451	2576	2372	2466	2500	**2580**	2014	2383
	exiv2	16529	17500	**19245**	14117	14417	16602	18952	7637	11437
JPG	jhead	1228	1258	**1281**	1244	1244	1244	1253	856	1246
PCAP	tcpdump	16772	19201	**22963**	14535	13743	19823	22930	1561	11837
PNG	pngtest	3219	3223	3629	3217	3239	3268	3617	2846	**4802**
TIFF	tiffdump*	1145	1155	**1177**	1054	1073	1104	1135	522	1128
TTF	freetype_parser*	9893	**10520**	10370	6362	6309	10030	7278	4513	7316
WAV	sfinfo	2477	2501	**2632**	2326	2326	2379	2498	2019	2566
XLS	xls2csv	2476	2699	**2706**	2512	2510	2619	2424	1841	2470
ZIP	7za	24696	25266	**28159**	21429	22089	25226	27979	13299	21833

[1] * for trained pairs of formats and programs.
[2] B for Field Boundary; T for Field Type; P for Power Scheduling Algorithm.

probing stage to infer the field boundary and field type for improving fuzzing efficiency. TIFF-fuzzer and WEIZZ use format knowledge to increase fuzzing efficiency.

5.3.1 Code Coverage Result

For each target program, we run all fuzzers for 24 hours with 5 repetitions. Then we measure the average BB coverage instead of path coverage since not all the fuzzers use the same metric to calculate the path.

We can draw the following conclusions from the result in Table 8. First, AIFORE increases the coverage significantly for most targets except `pngtest` for which WEIZZ performs the best. The reason is that WEIZZ can not only detect the `checksum` field in the file, but it can also correct the `checksum` values. However, AIFORE does not support `checksum` value correction even though it can be aware that the field is a `checksum` field. For all the targets, AIFORE (B+T+P) (i.e., enable the field boundary, type analysis, and utilize the power scheduling algorithm) has an average 6% and 26% increment on average compared with ProFuzzer and WEIZZ respectively, as shown in Figure 7.

Second, the coverage increment with AIFORE is significant, even though the target program and the file format are unseen.

Third, AIFORE achieves the best performance. For TIFF-fuzzer, since it aims at maximizing the likelihood of triggering a bug, we find it is not good at increasing the code coverage. Although ProFuzzer also performs better than format-unaware fuzzers like AFL and AFLFast in most cases, its analysis time is proportional to the input size, which makes it not scaleable to large inputs. Observe that ProFuzzer performs the worst in `XLS` except TIFF-fuzzer. The reason is that the minimum size of the `XLS` seed is above 1k bytes which is too large for ProFuzzer.

5.3.2 Bugs Found by AIFORE

Coordinated Vulnerability Disclosure. To explore AIFORE's ability of bug finding, we fuzz the real-world

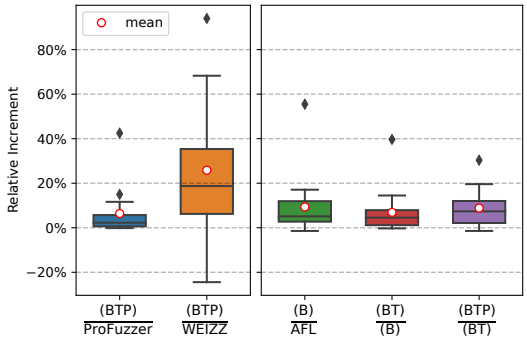


Figure 7: Coverage comparison between AIFORE and other SOTA fuzzers, and coverage increment by each module of AIFORE.

programs from Table 4 for 7 days. With the help of format knowledge, AIFORE finds 34 bugs (20 are uncovered by other fuzzers) in total after manual deduplication, including 10 buffer overflow bugs (CWE-122), 18 NULL pointer dereference bugs (CWE-476), and 6 double-free bugs (CWE-617). Among the 20 bugs that are uncovered by other fuzzers, 18 of them have been known to vendors before. AIFORE finds 2 vulnerabilities in the newest version of `xls2csv`, and we have responsibly reported both bugs to the Launchpad, and the track IDs are 1901462 and 1901463[5]. Here we analyze 2 of 34 bugs to illustrate how AIFORE finds them.

Sfinfo. `Sfinfo` is a program for parsing `WAV` files and showing their properties or metadata to users. AIFORE reports a heap overflow bug during the testing. The bug locates in a function for parsing an enumeration field called `FormatTag`. The field represents the way the wave data is stored and affects how the following bytes will be processed. If the `FormatTag` field in the input file is mutated to `WAVE_FORMAT_ADPCM`, then the length data might be calculated incorrectly, which then causes a heap overflow. The key to triggering this bug is

[5] https://launchpad.net/ The bugs have not been opened to the public when we submit the paper.

setting `FormatTag` equal to `WAVE_FORMAT_ADPCM`, which is easier to achieve if the fuzzer is aware of the boundary and type of this field. The `FormatTag` is an enumeration field and AIFORE identifies the boundary and the type of this field correctly. Then it mutates the field accordingly and triggers this bug.

Readelf. During the testing of `readelf`, AIFORE finds a stack overflow bug. The bug can be triggered when a 4-bytes `offset` field is mutated to an invalid value. The `offset` field serves as indicating the location of a string. When the string's size is bigger than 256 bytes. a call of `sprintf` will cause the stack buffer overflow. The root cause of this bug can be concluded as (1) the `offset` field must be set to point to an invalid string region, and (2) the string field pointed to must be long enough to trigger the stack overflow. We backtrack the log and find that AIFORE identifies the 4-bytes `offset` field correctly, and mutates the field as a whole to an invalid value. The invalid offset points to a string field and AIFORE mutates the string field with a longer length. So it can trigger the bug.

5.4 RQ4: Contribution of Each Module

In this section, we analyze how each module of AIFORE contributes to the fuzzing performance. We investigate the BBs covered by different components during fuzzing in §5.3. The result is shown from column 3 to column 5 in Table 8, and we draw a boxplot in Figure 7. We have the following conclusions. First, the module of boundary identification helps AIFORE to mutate the bytes which have the same semantic and belongs to one field, and it boosts AIFORE to cover 9.3% more BBs than AFL. Besides, AIFORE uses the AI model to predict the field type, which helps the fuzzer know how to mutate. Benefiting from the field type prediction module, AIFORE increases another 6.9% code coverage on average for all the targets. At last, AIFORE utilizes a novel power scheduling algorithm to help the fuzzer assign more energy to those formats which are not mutated adequately. It brings an 8.8% increment in coverage. Different from AFLSmart, our approach does not rely on the constraints in the `pit` file. We further analyze how it works with a case study in Appendix §B.3.

6 Related Work

6.1 Format-Aware Fuzzing

Format-aware fuzzers try to understand the format of inputs to increase the fuzzing efficiency. TIFF-fuzzer [6] performs bug-directed mutation by inferring some program variable types (e.g., `int`, `char*`) of input fields. ProFuzzer [8] uses predefined rules to deduce fields and corresponding types. However, adding more data types is labor-intensive. Besides, the rules may not be accurate enough to cover all the cases. Steelix [34] identifies magic numbers in the input to pass value validation. But it does not analyze fields of other types. Intriguer [2] utilizes light-weighted taint analysis to find multi-byte fields processed by instructions in the program, then uses the field-level knowledge to optimize symbolic execution. From the perspective of file format, it can only extract a small part of the fields because of its incomplete traces. WEIZZ [9] splits the input into fields according to dependencies between input bytes and comparison instructions, which ignores the bytes not affecting the control flow of the program.

AIFORE extracts more accurate, concrete, and complete format knowledge of field boundaries and semantic types, which improves fuzzing. Besides, AIFORE utilizes a novel power scheduling algorithm to balance the power for different formats.

6.2 Input Format Reverse Engineering

Input format reverse engineering works can be classified into two main categories regarding the problem they solve.

Field Boundary Identification. Identifying the boundary of different fields in the input is fundamental to reversing the format. Several works try to split the input into fields based on taint analysis [3, 4, 35–38] or trace analysis of a few network messages [5, 30, 39–41]. The closest work to AIFORE is Tupni [4], which uses the weighted taint information in the instruction, as long as a greedy algorithm to identify different fields. However, the instruction-level taint information may produce false positives since it does not consider the semantic. Besides, Tupni [4] is a coarse-grained method in which the record it identifies may contain several fields, rather than a single field. AIFORE considers the BB as the minimum functional unit instead of the instruction. AutoFormat [35] combines dynamic taint analysis and call stack to build a field tree. It relies much on the tokenization and the operation of the tree itself, rather than on how the program processes different fields. MIMID [42] and AUTOGRAM [43] also rely on dynamic taint analysis and call stack analysis. They are used to extract context-free grammar for text-based inputs rather than context-sensitive ones like binary-based inputs which are more complex. Reverx [44] splits the input into fields by predefined delimiters and it cannot be used for binary messages. There are some other works [30, 45] that try to identify the field by analyzing a large number of high-quality inputs. Nevertheless, AIFORE can extract the field knowledge with only one input.

Field Type Identification. Identifying the type of distinct fields is also an important problem. Current works [3, 34, 46, 47] mainly utilize dynamic program analysis and predefined rules to classify the input field into types. Dispatcher [48] identifies the field type by leveraging taint analysis and heuristics rules similar to TIFF-fuzzer. Polyglot [3] tries to identify the keywords and separators in the protocol message. It also tries to identify the length field by heuristics rules. However, the

identified field types are limited, and the heuristic rules have limitations when the message is not strict. For example, a protocol may allow multiple delimiters.

Different from the existing works, AIFORE utilizes a machine-learning model to extract the feature and predicts the field types automatically. We feed multi-dimension semantic features from the tainted instructions, format strings, and library calls in related BBs to train the model. In this way, AIFORE does not require extra manually-defined rules and therefore it is more general.

6.3 Binary Analysis with AI

The closest works in AI-based binary analysis include binary similarity detection [23, 49] and semantic information recovery [50, 51]. For the binary similarity detection, [23] utilizes BERT and CNN to find similar code. [52] uses Structure2vec to vectorize the CFGs. Different from existing works, AIFORE aims at classifying the input fields into different categories, which is a classification problem instead of an embedding problem. There are also some works [50, 51] that try to recover the semantic information (e.g., function names, variable types) from the binary program with AI. However, the program variable type is simpler than the field type. For example, a variable of `int` type may represent an `offset` or a `size` field. AIFORE aims to recover the semantic type of an input field, which is harder but more useful.

7 Limitation

While AIFORE has good accuracy for field boundary detection and type analysis, some limitations remain. As described, our method fundamentally relies on dynamic taint analysis. Thus, the key limitation of AIFORE is that how the program parses the input highly affects the result. For example, if a program does not parse some fields, AIFORE cannot extract the format knowledge. Further, if a program parses individual bytes of a field separately, AIFORE may produce false positives. However, we can overcome this issue by feeding the input to several programs that can parse this format. The second limitation is that our analysis runs at byte granularity, which means bit-level fields cannot be analyzed. Supporting bit-level analysis is technically possible but requires further engineering and optimization. The byte-level analysis also indicates that AIFORE does not support text-based input. The minimum unit of text-based input is a keyword rather than a byte. Third, it is hard for AIFORE to analyze the cases of input encryption or code obfuscation (e.g., in malware or ransomware). There is no obvious format information in encrypted input, and the developer may also use obfuscation code to hide the operation pattern to parse the file. There are several orthogonal works, for example, Reformat [38] that try to reverse the format of encrypted input.

8 Conclusion

Input format knowledge is useful for fuzzing to discover vulnerabilities in programs. Existing approaches struggle at correctly recognizing or applying the input format. We introduce AIFORE to automatically reverse engineer input format and later guide the fuzzing process. Specifically, we propose to utilize taint analysis to infer basic blocks responsible for processing each input byte, and group input bytes with a minimum cluster algorithm. Further, we utilize a neural network to infer the type of input fields based on the behavior of basic blocks. Based on the input knowledge, we present a novel power scheduling algorithm for fuzzers. A systematic evaluation shows that this solution has better effectiveness and efficiency than existing baselines.

Acknowledgements

This work was supported by the National Key Research and Development Program of China (2021YFB2701000), National Natural Science Foundation of China (61972224), Beijing National Research Center for Information Science and Technology (BNRist) under Grant BNR2022RC01006.

References

[1] J. Chen, J. Jiang, H. Duan, N. Weaver, T. Wan, and V. Paxson, "Host of Troubles: Multiple Host Ambiguities in HTTP Implementations," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. Vienna Austria: ACM, Oct. 2016, pp. 1516–1527.

[2] M. Cho, S. Kim, and T. Kwon, "Intriguer: Field-level constraint solving for hybrid fuzzing," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 515–530.

[3] J. Caballero, H. Yin, Z. Liang, and D. Song, "Polyglot: Automatic extraction of protocol message format using dynamic binary analysis," in *Proceedings of the 14th ACM conference on Computer and communications security*, 2007, pp. 317–329.

[4] W. Cui, M. Peinado, K. Chen, H. J. Wang, and L. Irun-Briz, "Tupni: Automatic reverse engineering of input formats," in *Proceedings of the 15th ACM conference on Computer and communications security*, 2008, pp. 391–402.

[5] W. Cui, J. Kannan, and H. J. Wang, "Discoverer: Automatic protocol reverse engineering from network traces." in *USENIX Security Symposium*, 2007, pp. 1–14.

[6] V. Jain, S. Rawat, C. Giuffrida, and H. Bos, "Tiff: using input type inference to improve fuzzing," in *Proceedings of the 34th Annual Computer Security Applications Conference*, 2018, pp. 505–517.

[7] Z. Lin, X. Zhang, and D. Xu, "Automatic reverse engineering of data structures from binary execution," in *Proceedings of the 11th Annual Information Security Symposium*, 2010, pp. 1–1.

[8] W. You, X. Wang, S. Ma, J. Huang, X. Zhang, X. Wang, and B. Liang, "Profuzzer: On-the-fly input type probing for better zero-day vulnerability discovery," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 769–786.

[9] A. Fioraldi, D. C. D'Elia, and E. Coppa, "Weizz: Automatic grey-box fuzzing for structured binary formats," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2020, pp. 1–13.

[10] V. Pham, M. Böhme, A. E. Santosa, A. R. Caciulescu, and A. Roychoudhury, "Smart greybox fuzzing," *IEEE Transactions on Software Engineering*, 2019.

[11] "The peach project," https://www.peach.tech/.

[12] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Advances in neural information processing systems*, vol. 25, pp. 1097–1105, 2012.

[13] T. Yue, P. Wang, Y. Tang, E. Wang, B. Yu, K. Lu, and X. Zhou, "{EcoFuzz}: Adaptive {Energy-Saving} greybox fuzzing as a variant of the adversarial {Multi-Armed} bandit," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 2307–2324.

[14] "Automatically inferring file syntax with afl-analyze," 2016, https://lcamtuf.blogspot.com/2016/02/say-hello-to-afl-analyze.html.

[15] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as markov chain," *IEEE Transactions on Software Engineering*, vol. 45, no. 5, pp. 489–506, 2017.

[16] M. I. Jordan and T. M. Mitchell, "Machine learning: Trends, perspectives, and prospects," *Science*, vol. 349, no. 6245, pp. 255–260, 2015.

[17] J. Wang, Y. Yang, J. Mao, Z. Huang, C. Huang, and W. Xu, "Cnn-rnn: A unified framework for multi-label image classification," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 2285–2294.

[18] J. Lee, T. Avgerinos, and D. Brumley, "Tie: Principled reverse engineering of types in binary programs," 2011.

[19] A. Slowinska, T. Stancescu, and H. Bos, "Howard: A dynamic excavator for reverse engineering data structures." in *NDSS*, 2011.

[20] "Tool interface standard (tis) executable and linking format (elf) specificatione," 1995, https://refspecs.linuxfoundation.org/elf/elf.pdf.

[21] Y. Shoshitaishvili, R. Wang, C. Hauser, C. Kruegel, and G. Vigna, "Vex," https://github.com/angr/pyvex.

[22] H. Lee and H. Kwon, "Going deeper with contextual cnn for hyperspectral image classification," *IEEE Transactions on Image Processing*, vol. 26, no. 10, pp. 4843–4855, 2017.

[23] Z. Yu, R. Cao, Q. Tang, S. Nie, J. Huang, and S. Wu, "Order matters: Semantic-aware neural networks for binary code similarity detection," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 34, no. 01, 2020, pp. 1145–1152.

[24] A. Rebert, S. K. Cha, T. Avgerinos, J. Foote, D. Warren, G. Grieco, and D. Brumley, "Optimizing seed selection for fuzzing," in *23rd USENIX Security Symposium (USENIX Security 14)*. San Diego, CA: USENIX Association, Aug. 2014, pp. 861–875. [Online]. Available: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/rebert

[25] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, "Vuzzer: Application-aware evolutionary fuzzing." in *NDSS*, vol. 17, 2017, pp. 1–14.

[26] V. P. Kemerlis, G. Portokalidis, K. Jee, and A. D. Keromytis, "libdft: Practical dynamic data flow tracking for commodity systems," in *Proceedings of the 8th ACM SIGPLAN/SIGOPS conference on Virtual Execution Environments*, 2012, pp. 121–132.

[27] F. Wang and Y. Shoshitaishvili, "Angr-the next generation of binary analysis," in *2017 IEEE Cybersecurity Development (SecDev)*. IEEE, 2017, pp. 8–9.

[28] Lonami, "Autoit scripting language," 2018, https://www.autoitscript.com/site/.

[29] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, "Evaluating fuzz testing," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 2123–2138.

[30] M. A. Beddoe, "Network protocol analysis using bioinformatics algorithms," *Toorcon*, 2004.

[31] A. Orebaugh, G. Ramirez, and J. Beale, *Wireshark & Ethereal network protocol analyzer toolkit*. Elsevier, 2006.

[32] D. She, A. Shah, and S. Jana, "Effective Seed Scheduling for Fuzzing with Graph Centrality Analysis," in *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, Apr. 2022, pp. 1558–1558.

[33] M. Zalewski, "American fuzzy lop," http://lcamtuf.coredump.cx/afl/, 2014.

[34] Y. Li, B. Chen, M. Chandramohan, S.-W. Lin, Y. Liu, and A. Tiu, "Steelix: program-state based binary fuzzing," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, 2017, pp. 627–637.

[35] Z. Lin, X. Jiang, D. Xu, and X. Zhang, "Automatic protocol format reverse engineering through context-aware monitored execution." in *NDSS*, vol. 8. Citeseer, 2008, pp. 1–15.

[36] P. M. Comparetti, G. Wondracek, C. Kruegel, and E. Kirda, "Prospex: Protocol specification extraction," in *2009 30th IEEE Symposium on Security and Privacy*. IEEE, 2009, pp. 110–125.

[37] B. Cui, F. Wang, T. Guo, G. Dong, and B. Zhao, "Flowwalker: a fast and precise off-line taint analysis framework," in *2013 Fourth International Conference on Emerging Intelligent Data and Web Technologies*. IEEE, 2013, pp. 583–588.

[38] Z. Wang, X. Jiang, W. Cui, X. Wang, and M. Grace, "Reformat: Automatic reverse engineering of encrypted messages," in *European Symposium on Research in Computer Security*. Springer, 2009, pp. 200–215.

[39] S. Kleber, H. Kopp, and F. Kargl, "{NEMESYS}: Network message syntax reverse engineering by analysis of the intrinsic structure of individual messages," in *12th {USENIX} Workshop on Offensive Technologies ({WOOT} 18)*, 2018.

[40] I. Bermudez, A. Tongaonkar, M. Iliofotou, M. Mellia, and M. M. Munafò, "Towards automatic protocol field inference," *Computer Communications*, vol. 84, pp. 40–51, 2016.

[41] J. Kannan, J. Jung, V. Paxson, and C. E. Koksal, "Semi-automated discovery of application session structure," in *Proceedings of the 6th ACM SIGCOMM conference on Internet measurement*, 2006, pp. 119–132.

[42] R. Gopinath, B. Mathis, and A. Zeller, "Mining input grammars from dynamic control flow," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2020. New York, NY, USA: Association for Computing Machinery, 2020, pp. 172–183.

[43] M. Höschele and A. Zeller, "Mining input grammars with AUTOGRAM," in *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, 2017, pp. 31–34.

[44] J. Antunes, N. Neves, and P. Verissimo, "Reverx: Reverse engineering of protocols," 2011.

[45] I. Bermudez, A. Tongaonkar, M. Iliofotou, M. Mellia, and M. M. Munafo, "Automatic protocol field inference for deeper protocol understanding," in *2015 IFIP Networking Conference (IFIP Networking)*. IEEE, 2015, pp. 1–9.

[46] T. Wang, T. Wei, G. Gu, and W. Zou, "Taintscope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection," in *2010 IEEE Symposium on Security and Privacy*. IEEE, 2010, pp. 497–512.

[47] Z. Lin and X. Zhang, "Deriving input syntactic structure from execution," in *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, 2008, pp. 83–93.

[48] J. Caballero, P. Poosankam, C. Kreibich, and D. Song, "Dispatcher: Enabling active botnet infiltration using automatic protocol reverse-engineering," in *Proceedings of the 16th ACM conference on Computer and communications security*, 2009, pp. 621–634.

[49] F. Zuo, X. Li, P. Young, L. Luo, Q. Zeng, and Z. Zhang, "Neural machine translation inspired binary code similarity comparison beyond function pairs," *arXiv preprint arXiv:1808.04706*, 2018.

[50] J. He, P. Ivanov, P. Tsankov, V. Raychev, and M. Vechev, "Debin: Predicting debug information in stripped binaries," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 1667–1680.

[51] Y. David, U. Alon, and E. Yahav, "Neural reverse engineering of stripped binaries using augmented control flow graphs," *Proceedings of the ACM on Programming Languages*, vol. 4, no. OOPSLA, pp. 1–28, 2020.

[52] X. Xu, C. Liu, Q. Feng, H. Yin, L. Song, and D. Song, "Neural network-based graph embedding for cross-platform binary code similarity detection," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 363–376.

[53] P. V. Mockapetris, "Rfc1035: Domain names-implementation and specification," 1987.

[54] R. R. Selvaraju, M. Cogswell, A. Das, R. Vedantam, D. Parikh, and D. Batra, "Grad-cam: Visual explanations from deep networks via gradient-based localization," in *Proceedings of the IEEE international conference on computer vision*, 2017, pp. 618–626.

Appendices

A Comparison of Protocol Reverse Engineering

Beyond the file inputs, we also choose two protocols, `ARP` and `DNS`, to test and compare the result with Wireshark [31], Polyglot [3], and PI [30]. The program we choose to parse `ARP` is `arping` which is provided by the system of Ubuntu 18.04, and the parameter is `arping 192.168.1.1 -I eno2 -c 1`. This command produces an `ARP` request and then parses an `ARP` reply packet. The program to parse `DNS` is `nslookup` from BIND 9, and the parameter we use is `nslookup example.com`.

In this experiment, we only focus on reversing the format of the response part (i.e., `ARP` reply and `DNS` response packet). Given only one response packet, AIFORE can successfully reverse the format via taint analysis and related analysis. However, during the test against PI, it fails to reverse the format given only one response packet, so we change the testing command from `-c 1` to `-c 10` to collect 10 response packets for PI to analyze. During the execution of the command, we capture the packets so that we can later compare the results.

We use the format knowledge taken from Wireshark as the ground truth. Wireshark identifies protocol formats by template scripts written carefully by community experts. However, we find AIFORE can extract more detailed format knowledge in some cases, like the `DNS` target. Thus, we also use the RFC 1035 [53], which defines the `DNS` protocol format specification as complementary to the ground truth. The orange parts represent the wrong results. AIFORE can not only identify the field boundaries but also identify the field types. AIFORE predicts the field type correctly for 4 fields out of 5.

The results of `DNS` and `ARP` format reverse engineering are shown in Figure 8 and Figure 9 respectively. The orange parts represent the wrong results. From the result, we can learn that AIFORE extracts more detailed and correct information than Wireshark in the green parts of DNS. From the RFC specification [53] of `DNS`, we know that the `QNAME` field contains several `labels`, and each label consists of a byte representing the length followed by a number of octets representing data. Wireshark marks the `QNAME` as a whole and is not able to split it into individual labels. AIFORE extracts more detailed field boundaries than Wireshark, as shown in the result. AIFORE also identifies the field type correctly.

B Case Study

In this section, we analyze some concrete examples to help understand how AIFORE outperforms other state-of-art works from the view of field boundary recognition and field type classification. We take the `ELF` as an example to illustrate.

B.1 Field Boundary Case

We take 2 format-aware fuzzing tools that can identify fields in the input as examples. The field extraction result during fuzzing is shown in Figure 10.

From the result, TIFF-fuzzer and AIFORE split the first four bytes (i.e., `magic_number` field) into single-byte fields. However, since the program parses the bytes one by one, then it is better to fuzz each of the bytes rather than as a whole. For WEIZZ, there are a few false negatives. The reason is that WEIZZ relies on `cmp` instruction when extracting the fields, which is not sufficient.

In AIFORE, we perform a complete taint analysis on valuable inputs, rather than WEIZZ, which achieves a higher accuracy on field identification and thus can increase the fuzzing efficiency better than other fuzzers, as shown in Table 8.

B.2 Field Type Case

For field type identification, state-of-art works generally depend on specific rules, and the field types they identified are usually program types rather than the semantic type of a field. For example, TIFF-fuzzer infers field types based on APIs (`strcmp`,`strcpy`) in libraries and then splits the field into several program variable types such as `char*` and `int`. However, such rules and types may not be sufficient. Take the section name, `s_name`, in the `ELF` file as an example. Such fields are `string` type. However, TIFF-fuzzer considers them as consecutive `int` bytes. The reason is that `readelf` uses `repe cmpsb` instruction rather than `strcmp` call to parse `s_name`. In AIFORE, it marks this field as a magic number, which is also reasonable, since the program tries to compare the section name with hardcoded strings. We then investigate

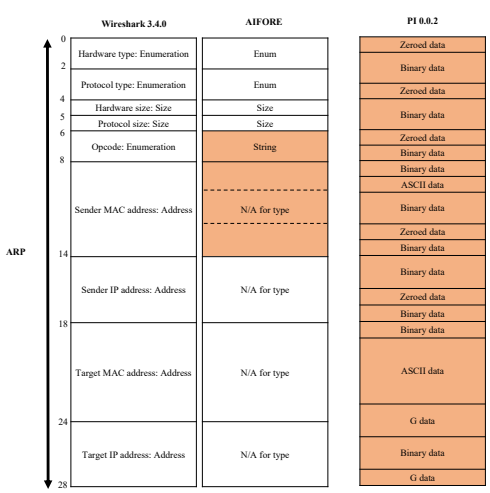


Figure 8: Results of the ARP reverse engineering.

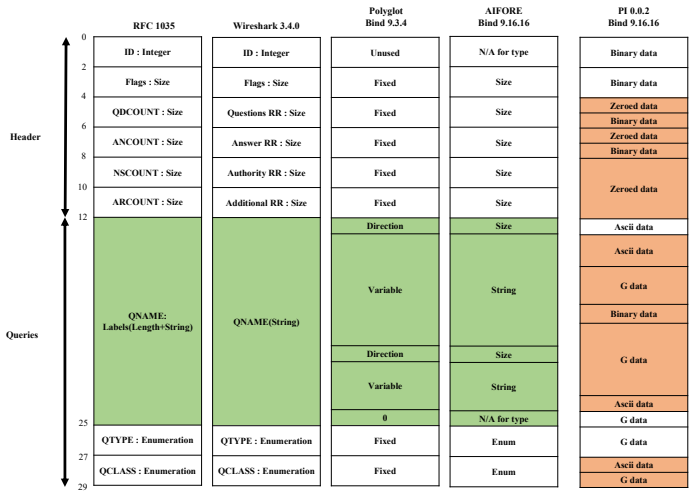
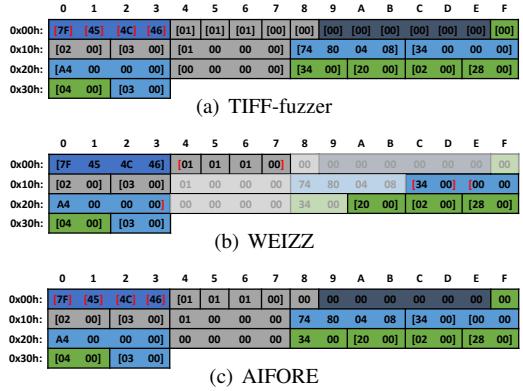


Figure 9: Results of the DNS format reverse engineering.



(a) TIFF-fuzzer

(b) WEIZZ

(c) AIFORE

Figure 10: Field identification results. Bytes in red brackets indicate they are different than the specification, while the shallow parts indicate the target fails to extract the fields.

why the model in AIFORE predicts the field as a magic number. We leverage the Grad-cam [54], which is used to explain why a model makes a specific decision. It helps humans to understand the internal working principle of a classification model.

To explain the decision of the model, we feed the vectorized semantic feature (IR operation, library call, and format strings) of the `magic` field to Grad-cam. Then we observe which feature plays the most important role in the decision. We choose the top 5 features of Grad-cam to observe: **['CmpLT32U', '128to64', 'CmpLE64S', 'And8', 'CmpLE32U']**. As the result shows, the `cmp` feature plays the most important role when the model predicts the field as a magic number, which is reasonable.

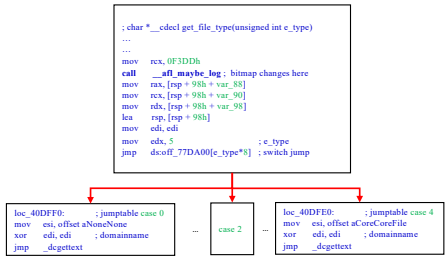


Figure 11: ProFuzzer failure case.

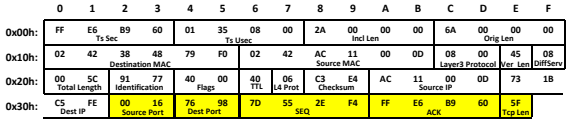
Another work that can identify the semantic type of a field is ProFuzzer. We take the field `e_type` locates at offset 0x10 in Figure 10 as an example. It represents the file type (e.g., ET_EXEC), which is an enumeration. During the probing stage, ProFuzzer considers it an incorrect `offset` type. We then investigate the reason and find this is due to the limitation of code coverage bitmap. ProFuzzer mutates each of the bytes in `e_type` to observe the similarity of the bitmaps. But from Figure 11, we can learn that different switches share the same bitmap, which leads ProFuzzer to make the wrong decision. However, in AIFORE, it predicts the field type based on how the program parses the specific field, and we also use forward slicing to merge BB to get more code features, which makes it able to recognize this field correctly.

B.3 Power Scheduling Case Study

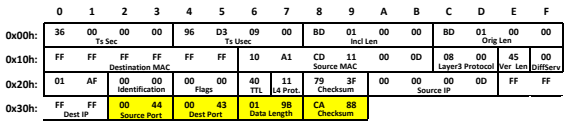
To better understand how the power scheduling works in AIFORE, we choose `tcpdump` as a target to answer two questions. First, do valuable seeds really bring new formats? Second, can AIFORE assign more power to those formats which are mutated less?

`tcpdump` is a program to parse `pcap` files. The `pcap` file consists of several data structures (e.g., pcap header, frame) which are composed of several fields. Before analyzing the

result, let us illustrate the frame structure first. As shown in Figure 12, the frame has some fields with consecutive bytes (e.g., `Incl Len` consisting of bytes at offsets from 0x08 to 0x0B) and some fields with single-byte (e.g., `L4 Prot`) values. Take `L4 Prot` at offset 0x27 as an example, it indicates the data type (TCP, UDP, etc.) of Layer 4, which is highlighted in yellow. `tcpdump` will dispatch different codes to parse various kinds of L4 layer data.



(a) Frame with TCP packet.

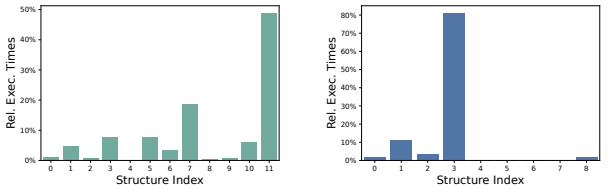


(b) Frame with UDP packet.

Figure 12: Frame structure definition.

To answer the first question, we investigate how many valuable seeds bring new structures. We consider the seeds valuable if they can reach more BBs (e.g., 3%) than others. We run AIFORE to fuzz `tcpdump` for 5 hours and investigate how many new structures are produced. The result is shown in Table 9. From the result, we have 30 valuable seeds in total, and for each of the seeds, if it brings new structures, we then count the number. For example, there are 11 seeds bring new format variants of Layer 2 data. In summary, there are 20 valuable seeds that produce new structures out of 30.

To further understand the performance of the power scheduling in AIFORE, we also record the execution times for each of the new structures during fuzzing. We have two groups of experiments. The only difference is that the first group is equipped with the power scheduling algorithm, while the other is not. We feed `tcpdump` with the same seeds, and we fuzz them for 3 hours with AIFORE. The result is shown in Figure 13. As it can be seen, if AIFORE is equipped with power scheduling, some of the fuzzing power can be re-assigned to those new structures, which are mutated less. However, the fuzzer focus most of its power on one structure without the power scheduling. Noted that there is a structure that is mutated with large execution times in both of the groups. The reason is that the power scheduling starts to work after the fuzzer gets to run for a while, rather than at the beginning.



(a) Power scheduling equipped. (b) Power scheduling unequipped.

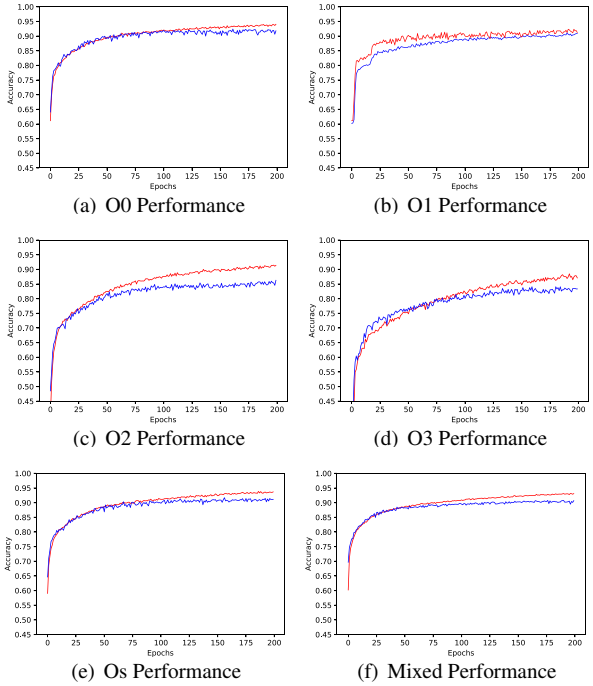
Figure 13: Execution times analysis for power scheduling algorithm.

Table 9: New formats produced by valuable seeds.

	Layer2	Layer3	Layer4	No Variant
Origin Seed Id	88, 267, 287, 296, 310, 360, 368, 549, 618, 725, 765	15, 207, 713, 728, 759, 784, 1928	306, 593	27, 51, 300, 420, 526, 551, 594, 924, 937, 1785
Total	11	7	2	10

C Other Tables and Figures

Figure 14 shows the field type accuracy results of models trained on programs with different compiler optimization levels. The detailed explanation is in Question 1 of §5.1.2.



(a) O0 Performance (b) O1 Performance

(c) O2 Performance (d) O3 Performance

(e) Os Performance (f) Mixed Performance

Figure 14: Field type accuracy of models trained on different programs (red/blue lines: accuracy on training/validation set).