NestFuzz: Enhancing Fuzzing with Comprehensive Understanding of Input Processing Logic

Peng Deng
Fudan University
pdeng21@m.fudan.edu.cn

Zhemin Yang
Fudan University
yangzhemin@fudan.edu.cn

Lei Zhang
Fudan University
zxl@fudan.edu.cn

Guangliang Yang
Fudan University
yanggl@fudan.edu.cn

Wenzheng Hong
Fudan University
wzhong20@fudan.edu.cn

Yuan Zhang
Fudan University
yuanxzhang@fudan.edu.cn

Min Yang
Fudan University
m_yang@fudan.edu.cn

ABSTRACT

Fuzzing is one of the most popular and practical techniques for security analysis. In this work, we aim to address the critical problem of high-quality input generation with a novel input-aware fuzzing approach called NestFuzz. NestFuzz can universally and automatically model input format specifications and generate valid input.

The key observation behind NestFuzz is that the code semantics of the target program always highly imply the required input formats. Hence, NestFuzz applies fine-grained program analysis to understand the input processing logic, especially the dependencies across different input fields and substructures. To this end, we design a novel data structure, namely *Input Processing Tree*, and a new cascading dependency-aware mutation strategy to drive the fuzzing.

Our evaluation of 20 intensively-tested popular programs shows that NestFuzz is effective and practical. In comparison with the state-of-the-art fuzzers (AFL, AFLFast, AFL++, MOpt, AFLSmart, WEIZZ, ProFuzzer, and TIFF), NestFuzz achieves outperformance in terms of both code coverage and security vulnerability detection. NestFuzz finds 46 vulnerabilities that are both unique and serious. Until the moment this paper is written, 39 have been confirmed and 37 have been assigned with CVE-ids.

CCS CONCEPTS

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

KEYWORDS

Fuzzing; Structure-aware; Code Coverage; Vulnerability

CCS '23, November 26–30, 2023, Copenhagen, Denmark

ACM ISBN 979-8-4007-0050-7/23/11…$15.00
https://doi.org/10.1145/3576915.3623103

ACM Reference Format:
Peng Deng, Zhemin Yang, Lei Zhang, Guangliang Yang, Wenzheng Hong, Yuan Zhang, and Min Yang. 2023. NestFuzz: Enhancing Fuzzing with Comprehensive Understanding of Input Processing Logic. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security (CCS '23), November 26–30, 2023, Copenhagen, Denmark.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3576915.3623103

1 INTRODUCTION

Fuzzing is an effective software testing technique that has been broadly applied in exploring and vetting software security with continuous input generation. It has proven useful and powerful in discovering numerous perilous vulnerabilities [12, 41]. One key challenge faced in the fuzzing procedure is generating a large number of effective inputs, which can be accepted by the target program and help explore as many paths as possible and hunt the vulnerabilities hidden in the deep. In past years, barrelful fuzzing tools applied input-agnostic mutation or generation strategies and usually generated tremendous invalid test cases, which were often rejected early by shadow code, e.g., input sanitizer and validations.

For smart and effective fuzzing, it is critical to learn the knowledge of input formats and requirements, especially when targeting real-world programs (often characterized by nested and structured input). Consider a typical real example of a popular MP4 multimedia player, which accepts binary MP4 files as input and contains a zero-day heap-based buffer overflow vulnerability. As the MP4 input format specifications are complicated, Figure 1 partially simplifies a crucial input structure of MP4. We find inside MP4, its input fields are arranged and organized structurally and hierarchically. Specifically, the outermost *moof* structure groups the *type*, *length*, and *payload* input fields. This means, the interpretation of *payload* highly depends on its *type* and *length*. The *payload* field contains two *mfhd* and *traf* substructures (i.e., field group), while *traf* is nested with another two *tfhd* and *sdtp* substructures. These substructures include their own fields, such as *smp_cnt* and *smp_info* in *sdtp*. Please note that substructures may be freely combined and grouped. Considering there are more than 500 MP4 substructure types available in practice, the input format space is quite large. It is not easy to generate such a vulnerable input and find the vulnerability along with a much longer path.

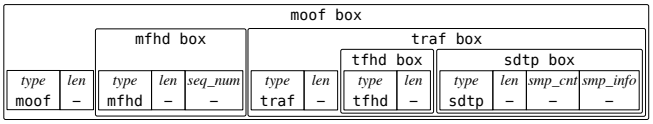


Figure 1: Simplified Partial Input Format of MP4, in which the *smp_info* field could carry the attack payload.

Generally, the in-depth understanding of input formats brings two important merits: (i) For path exploration, the knowledge of input formats can help cover as much code as possible by generating valid inputs, that satisfy the target program's input format specifications; (ii) For security exploitation, whose focus shifts to finding deep vulnerabilities, the understanding of input formats can guide the generation of ill-formed valuable input, helping hit the unexpected paths but still satisfy the input validations. However, achieving such a promising goal is not an easy task.

Recently, several fuzzing tools [17, 20, 31, 35, 43] were proposed and aimed to be aware of input formats. Nevertheless, they primarily focused on recognizing individual fields, while neglecting the hierarchy of input fields as well as substructures. In particular, ProFuzzer [43] and AIFORE [35] learned the boundaries and types of individual fields using statistical analysis or neural network models. However, they faced difficulties to obtain the landscape of the nested and structured input format. Their input mutation strategies conducted the changes in the single-field level, but hardly satisfied critical dependencies between different fields as well as hierarchy structure — for example, the generated *sdtp* substructure should be adjacent to a *tfhd* substructure, while these two substructures should be parts of another *traf* substructure (Figure 1). WEIZZ [17] primarily focused on the automatic identification of fields and chunks within chunk-based file formats, but neglected to take into account the inter-field dependencies and the input hierarchy structure. Furthermore, some other tools [7, 31] utilized pre-learned knowledge, i.e., input format template, that is manually defined based on the documentation of format specifications. On the one hand, many target programs (e.g., MP4 programs) have complicated input format specifications, thus requiring heavy manual efforts for format template construction. On the other hand, we observe these specifications frequently diverge from real-world code implementation, resulting in the inaccuracy of the format templates and impacting the detection outcomes.

In general, current format-aware fuzzing tools lacked a comprehensive understanding of input formats. This led to their input generation failing to meet program requirements, and was ineffective in exploring complex real-world programs, thereby failing to trigger deep vulnerabilities. For the purpose of verification, we have chosen several open-source state-of-the-art fuzzing tools, e.g., ProFuzzer [43] and AFLSmart [31], against the MP4 program, detailed in Section 2. Our analysis confirms existing tools faced challenges in learning the vulnerable input format (Figure 1) and generating harmful input. This is also verified in our evaluation on a board of data sets.

In this work, we propose a novel smart fuzzing approach, called NestFuzz, against the daunting problem of thoroughly learning the knowledge of input formats, and utilizing this knowledge to facilitate effective fuzzing. The main insight behind our NestFuzz approach is that the knowledge of input formats is generally implied by the interpretation and execution (referred to as *input processing logic*) of the tested program's input-accessing instructions, i.e., the instructions that depend on input data in terms of data-flow. Therefore, we can learn the input format knowledge of the target program, by analyzing and modeling its input processing logic. Beyond identifying individual fields in the input raw data, our modeling of input processing logic should additionally capture two important traits of input structure:

- **Inter-Field Dependency**. As an example of a field-level dependency, the *length* field of an MP4 *moof* substructure is used in a loop termination condition, for controlling the reading of its *payload* field, e.g., '*for (..; i < length; ..) buffer[i] = payload[i]*'. Once *length* is changed, the interpretation of *payload* will also be adjusted, i.e., its actual length should be equal to *length*. Hence, the *length* and *payload* fields should be grouped together.
- **Hierarchy Dependency**. Another more complicated feature is the inference of the substructure hierarchy. Correspondingly, programs may use recursive logic, e.g., loop and recursive function call, when executing input-accessing instructions, for reusing specific code logic of processing similar structures. For example, *moof*'s *payload* field consists of several different substructures, e.g., *traf*. Like *moof*, *traf* can similarly nest other substructures, they are parsed and dealt with by the same function.

Upon the above key insight, NestFuzz conducts smart fuzzing by 1) learning the knowledge of input formats from the interpretation and modeling of input processing logic, and 2) proposing and applying a novel knowledge-guided mutation strategy. Specifically, in the initial phase, NestFuzz first identifies the input-accessing instructions, by employing dynamic taint analysis on input raw data. An instruction involving a tainted operand is classified as an input-accessing instruction. Then, NestFuzz interprets and models the input processing logic from these collected input-accessing instructions. To achieve this goal, NestFuzz proposes a novel data structure, called *Input Processing Tree*, to accommodate and represent input processing logic. Importantly, *Input Processing Tree* can guide the modeling of input processing logic from several perspectives, including the dependencies of field-to-field and substructure-to-substructure.

In the second phase, based on *Input Processing Tree*, NestFuzz designs a novel cascading dependency-aware mutation strategy. In general, during path exploration, when NestFuzz mutates a part of the input seed's structure, it traverses the *Input Processing Tree* to find and adjust all other fields or substructures that depend (i.e., the inter-field and hierarchy dependencies) on this part, thus keeping the input structure remain valid. Furthermore, for security exploitation, NestFuzz prioritizes mutating newly-discovered and deeply-nested substructures to trigger the deep vulnerabilities. Since the deeply-nested substructures correspond to the deep processing logic in the program, prioritizing mutating these substructures can help trigger the deep bugs while ensuring the shallow processing logic is not violated.

We implement our prototype of NestFuzz and evaluate its effectiveness using the benchmark of 20 intensively-tested real-world programs. Our benchmark comprehensively encompasses 12

frequently-used binary file formats and 6 non-binary file formats. As a result, NestFuzz unveiled 46 unique vulnerabilities. We have responsibly disclosed them to the corresponding developers. Up to now, 39 have been confirmed and 37 have been assigned with CVE-ids (detailed in Section 5.3). We compared NestFuzz with the state-of-the-art structure-aware fuzzers, including AFLSmart [31], WEIZZ [17], ProFuzzer [43] and TIFF [20], and structure-unaware but general fuzzers, including AFL [30], AFLFast [10], AFL++ [18], and MOpt [28], on the above benchmark. The results show that NestFuzz outperforms the state-of-the-art fuzzers in terms of both test coverage and security vulnerability detection. For example, in comparison to structure-aware tools, e.g., WEIZZ, our tool identifies 1.56 times more unique lines and 1.67 times more unique branches. Compared with structure-unware tools, e.g., AFL++, our tool identifies 1.3 times more unique lines and 1.31 times more unique branches.

In summary, our contributions are outlined as follows:

- In this work, we focus on the daunting problem of a comprehensive understanding of input processing logic, and propose a novel fuzzing technique, called NestFuzz[1].
- With the novel design of structural representations for input, NestFuzz can effectively model input structure, including the hierarchy of fields and substructures.
- NestFuzz applies a cascading dependency-aware mutation strategy to adequately explore the input space and the code space of the program.
- Our evaluation shows NestFuzz outperforms the state-of-the-art fuzzing tools in both program exploration and security verification and finds 46 unique vulnerabilities, 37 of which have been assigned with CVE IDs.

2 MOTIVATING EXAMPLE

In this section, we follow the discussion of Section 1 and detail the representative real-world example of the popular MP4 parser. We break up our key insight that the input processing logic implies the input format and also discuss the limitations faced by existing tools. The MP4 program contains a dangerous *zero-day* heap-based buffer overflow vulnerability. We have responsibly reported this new vulnerability to the program developers and actively helped them fix this issue. Up to now, the vulnerability has been confirmed by the program developers and issued with a CVE ID.

As discussed in Section 1, it is not easy to trigger the vulnerability. The input format of its exploit has been shown in Figure 1. Please note that the presented input format is automatically interpreted and extracted from code implementation, rather than the MP4 format specification documentation. This is mainly because the specification documentation has many inconsistencies with real-world code implementation. Figure 2 shows the simplified related code snippet. The key vulnerable instruction appears at line 9 in the function *parse_boxes_internal()*. This function is used to parse the raw MP4 file with a loop (lines 3-10) recursively dealing with various MP4 substructures that have similar input field layouts. It calls *box_parse_ex()* (line 4) to extract the content of each substructure. Inside *box_parse_ex()* (line 11 of box_funcs.c), each

[1]We will open-source the prototype of NestFuzz at https://github.com/fdu-sec/NestFuzz

```
/* File: src/isomedia/isom_intern.c */
1  GF_Err parse_boxes_internal(GF_ISOFile *mov, u32 *boxType) {
2      GF_Box *a;
3      while (bs_available(mov->bs)) {
4          e = box_parse_ex(&a, mov->bs, ...);
5          switch (a->type) {
6              case BOX_MOOF: //MOOF box
7                  TrackBox *traf = list_get(a->TrackList, ...);
8                  if (traf->sdtp) { //SDTP box in TRAF box
9                      //Heap Buffer Overflow
10 }   }   }   }   }   }
```

```
/* File: src/isomedia/box_funcs.c */
11 GF_Err box_parse_ex(GF_Box **outBox, GF_BitStream *bs,  ...) {
12     u32 type = gf_bs_read_u32(bs);
13     GF_Box *newBox = box_new_ex(type, ...);
14     newBox->registry->read_fn(newBox, bs); //=moof_read...
15     *outBox = newBox;
16 }
17 GF_Err box_array_read(GF_Box *parent, GF_BitStream *bs){
18     GF_Err e;
19     GF_Box *a = NULL;
20     while (parent->length>=8) {
21         e = box_parse_ex(&a, bs, ...);
22         ...
23 }   }
```

```
/* File: src/isomedia/box_code_base.c */
24 GF_Err moof_read(GF_Box *s, GF_BitStream *bs) { ❶
25     return box_array_read(s, bs);
26 }
27 GF_Err mfhd_read(GF_Box *s, GF_BitStream *bs) { ❷
28     ...
29 }
30 GF_Err traf_read(GF_Box *s, GF_BitStream *bs) { ❸
31     TrackBox *ptr = (TrackBox *)s;
32     GF_Err e = box_array_read(ptr, bs);
33     if (!ptr->tfhd) {
34         return GF_ISOM_INVALID_FILE;
35 }   }
36 GF_Err tfhd_read(GF_Box *s, GF_BitStream *bs) { ❹
37     ...
38 }
39 GF_Err sdtp_read(GF_Box *s, GF_BitStream *bs) { ❺
40     ...
41 }
```

Figure 2: Motivating Example.

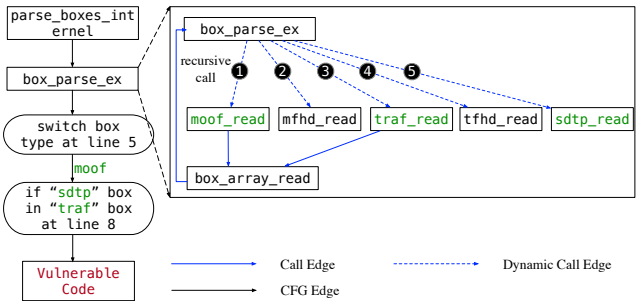


Figure 3: Abstract Input Processing Logic of the Motivating Example.

substructure, i.e., *newBox*, is parsed by the function pointer *read_fn*, which is pre-configured according to the substructure *type*. For example, a *moof*-type substructure is dealt with by the function *moof_read()* (at line 24), while *traf* is handled by *traf_read()* (at line 30).

For exploiting the heap vulnerability (line 9), the input data must satisfy several important constraints guarding the key vulnerable instruction. Figure 3 shows a pseudo-paradigm of the simplified input processing logic, indicating how the input is interpreted and validated. For example, the condition at line 6 requires the input *type* field must be BOX_MOOF. This determines that the outermost structure must be of *moof*. line 8 requires non-null *traf* and *sdtp* pointers. Thus, there should be a *traf* substructure under *moof* and another substructure *sdtp* under *traf*. line 33 needs a non-null *tfhd* pointer, determining that *traf* must have a *tfhd* substructure (otherwise the input will be rejected). In the *sdtp*, its *smp_info* field can carry the essential malicious attack payload.

This input processing logic discovers the hierarchy dependencies between substructures, and the demanded high-level structure of input data. In addition to these hierarchy dependencies, there are also dependencies between different fields. For instance, the *type* field (line 12) determines how to interpret its following data. Its value determines the function pointer *read_fn* at line 14. Consequentially, different substructure handlers will be called, e.g., calling *moof_read()* for the *moof*) substructure. Furthermore, the *length* field can determine the length of the *payload* field (line 20). Essentially, it not only determines the boundary of the *payload* single field, but also impacts the interpretation of the consecutive substructures, e.g., the start bytes of the next substructure adjacent the *payload*. Therefore, the inter-field dependencies should also be respected in input generation and mutation.

Table 1: The Number of Inputs Generated by Existing Fuzzing Tools for the Motivating Example.

Fuzzer	Input Num	❶	❶❷	❶❷❸	❶❷❸❹	❶❷❸❹❺
AFL [30]	1,436	65	1	0	0	0
AFLFast [10]	2,958	68	1	0	0	0
AFLSmart [31]	11,297	40	3	0	0	0
WEIZZ [17]	32,385	0	0	0	0	0
ProFuzzer [43]	9,688	8	0	0	0	0
TIFF [20]	99	0	0	0	0	0
NestFuzz	24,949	955	842	457	401	3

Applying Fuzzing. When a smart and effective fuzzer satisfies the inter-field and hierarchy dependencies, the called edges ❶❷❸❹❺ (Figure 3) should be traversed and executed sequentially. This means the input data is organized in a correct structure (Figure 1), and parsed recursively. However, when applying the state-of-the-art open-source input format-aware fuzzing techniques (e.g., AFLSmart and ProFuzzer) against this common example (with 24-hour timeout for each tool), we find existing tools produce numerous invalid inputs and face difficulties in constructing the required input structure (Figure 1). Table 1 presents the number of the generated inputs of each fuzzing tool, which can reach and enter the substructure handler functions (e.g., *read_fn()* at line 14). The input number greatly decreases as the input processing logic executes deeply. As a result, existing techniques are only able to explore up to the first two levels of nested logic, i.e., *moof_read()* and *mfhd_read()*, significantly hindering their effectiveness and practicality.

The reason is that previous work either did not satisfy the constraints of inter-field dependencies or hierarchy dependencies. The test cases they generate frequently destroy the initial valid input format, which is inadequate to trigger the deep vulnerability. For example, when a field, such as the *moof type* at lines 5 and 6, is changed and mutated to BOX_MOOF, its consecutive field content must be adjusted as well for totally conforming to the *moof's* structure. Since existing fuzzers will not introduce such an adjustment, the newly generated seeds commonly are invalid.

To tackle this problem, critical is learning the knowledge of input formats. As discussed above, this knowledge is implied by the input processing logic. Therefore, NestFuzz first understands the comprehensive structure (i.e., the inter-field and hierarchy dependencies) of input formats by modeling input processing logic, then uses the learned knowledge to guide the fuzzing. We present more details in the next two sections.

3 NESTFUZZ OVERVIEW

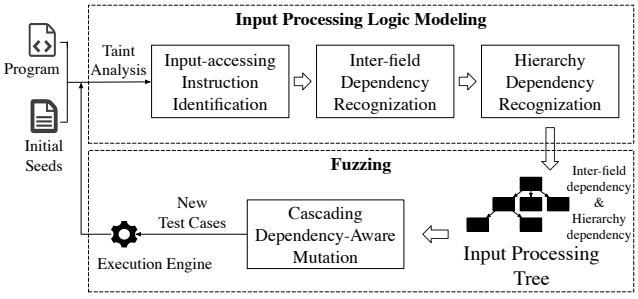


Figure 4: The Overall Architecture of NestFuzz.

In this work, we address the daunting problem of understanding and awareness of input formats, with proposing a new universal input format-aware fuzzing approach, named NestFuzz. As discussed in Section 1 and 2, the key intuition behind our NestFuzz approach is straightforward. Regardless of what specific input format a program requires, the code semantics of the target program, i.e., input processing logic, often accurately implies the knowledge of the demanded input format. Thus, generally, NestFuzz can first learn the knowledge of input formats by interpreting and modeling the input processing logic, and then leverage the obtained knowledge to guide effective fuzzing.

As shown in Figure 4, our NestFuzz approach mainly includes two phases. In the first phase of input processing logic modeling, NestFuzz first leverages taint analysis to identify input-accessing instructions. Then, NestFuzz recognizes the inter-field dependencies and hierarchy dependencies by understanding the control- and data-flow relationships between these input-accessing instructions. Last, NestFuzz proposes a novel data structure, namely *Input Processing Tree*, that can represent the whole structure of the input format.

In the second phase of fuzzing, NestFuzz designs a cascading dependency-aware mutation strategy. Based on the recognized dependencies, whenever NestFuzz mutates (field or structure-level) the input, it cascadingly mutates other affected fields or substructures to maintain the structure validity. Therefore, NestFuzz can continuously and effectively generate new high-quality test cases.

It is worth noting that the phase of input format modeling runs concurrently with the fuzzing procedure. The high-level process is presented in Algorithm 1. On the one hand, our scheduler consistently picks an input from our input queue, and then utilizes this input to drive the generation of *Input Processing Tree*. The fuzzer, on the other hand, continuously conducts a cascading mutation against the picked input based on *Input Processing Tree*, and performs dependency-aware fuzzing.

Algorithm 1 NestFuzz Fuzzing Procedures

```
Input: program, initial inputs
Output: a set of inputs
 1: Q = {initial inputs}
 2: S = Empty map
 3: procedure InputProcessingLogicModeling(program)
 4:     program_s = Instrument(program)              ▷ Algorithm 2
 5:     while not timeout do
 6:         input = Schedule(Q \ S.keys())
 7:         IPT = IPTAnalysis(program_s, input)
 8:         S[input] = IPT
 9: procedure Fuzz(program)
10:     program_c = AFLInstrument(program)           ▷ AFL instrumentation
11:     while not timeout do
12:         input = Select(Q)
13:         IPT = S[input]
14:         if IPT then
15:             p = AssignEnergy(input)
16:             for i from 1 to p do
17:                 input' = CascadingMutation(input, IPT)
18:                 run(program_c, input')
19:                 if input' has new coverage then
20:                     Q.add(input')
```

4 METHODOLOGY

In this section, we dive into the methodology of our NestFuzz approach with two essential analysis phases of modeling input processing logic and fuzzing with dependency-aware mutation.

4.1 Definitions

Below we define the terms frequently used in the remaining content.

Definition 1. A field F consists of a set of consecutive bytes. F usually has several properties, including its *start* and *end* byte position in raw data, its concrete *value*, and field *type*. Hence, F = $\{\bigcup_{start}^{end} byte_i, value, type\}$.

Definition 2. A substructure SF consists of a set of continuous fields or other substructures. Thus, $SF = \bigcup_{i,j} (F_i|SF_j)$.

Definition 3. A field F_1 may depend on another field F_2, if changing a property (e.g., value) of the field F_2 can affect the parsing of another field F_1. We use $D[F_1, F_2]$ to represent the dependency between F_1 and F_2.

4.2 Modeling Input Processing Logic

As discussed in Section 1 and 2, it is essential to analyze and model the input processing logic, achieving an in-depth understanding of the two crucial traits of input structure: the inter-field dependency and the hierarchy dependency. The input processing logic encompasses the instructions on how a program reads, interprets, and validates its inputs. Therefore, we take into account not only the syntax and format of the input data but also the control-flow and data-flow level dependencies between the instructions that process the input.

In this section, we detail our NestFuzz approach in the first phase of modeling the input processing logic. In general, NestFuzz first identifies the input-accessing instructions through dynamic taint analysis. Then, NestFuzz infers the inter-field dependencies and hierarchy dependencies respectively. Last, NestFuzz proposes the novel data stricture *Input Processing Tree* to accommodate the input processing logic. Below we present the details for each step.

4.2.1 Identifying Input-Accessing Instructions. Input-accessing instructions are the ones that involve input raw data. To identify input-accessing instructions, we utilize dynamic taint analysis on the input source and pinpoint instructions responsible for processing the input. Our taint analysis is conducted at the byte level based on DFSan [5]. In particular, we label all the input bytes as '*tainted*' and assign a tainted label '*shadow_value*' (e.g., offset in raw input data) to each input byte and trace their propagation. When a variable is assigned with taint labels, we can retrieve the input bytes flowing to the variable from the labels. Thus, when an instruction involves a tainted operand, it can be classified as an input-accessing instruction.

In this procedure, we focus on three types of input-accessing instructions, i.e., Load, Cmp, and Switch. We instrument these instructions to keep track of their instruction addresses and their operands' taint tags, which indicate the offsets of the corresponding input fields. Furthermore, an instruction may execute multiple times with different contexts and therefore different taint tags. We record each instance separately.

4.2.2 Recognizing Inter-field Dependencies. Input-accessing instructions provide rich code semantics, which can be utilized to infer the knowledge of input formats. In this step, we recognize key fields (including *length*, *offset*, *category*, and *payload*) and infer the dependencies between them. In general, our analysis mainly considers three types of inter-field dependencies that significantly impact the input structure, i.e., *D[payload, length]*, *D[payload, offset]* and *D[payload, category]*. It is worth noting that the *payload* field is often structured, nesting other fields or substructures. Below we present the details for each type of inter-field dependency.

• *D[payload, length]*: The *length* field can determine the length of another *payload* field. We recognize these 'length' and 'payload' fields and their dependency, based on the following code semantics. First, the *length* field may be used as a parameter of buffer-related system APIs. Use the following cases as an example. The parameter in the corresponding position is of the *length* type. Thus, we can further determine the *payload* variable and link them together, obtaining $D[payload, length]$. We model the system APIs that may have a tainted variable as its parameter. If a tainted variable is used as the corresponding parameter of some APIs that have a *length* type parameter, we recognize the taint source as a *length* type field. Additionally, from the API use, we can also infer the corresponding field of *payload*.

```
read(handler, buffer, length) // read content
buffer = malloc(length)    // allocate a buffer
memcpy(dst, src, length)   // copy from src to dst
```

Second, the *length* variable may be used in a loop condition to determine the iteration rounds of reading tainted data from input

to a buffer. As the following code shows, we can link the *length* and *buffer* variables together. Specifically, we recognize the tainted variable in the loop condition as the *length* field type, and all related tainted variables appeared in the loop body as parts of *payload*.

```
for (int i = 0; i < size; i++) {
    buffer[i] = input[i];
    ...
```

• *D[payload, offset]*: The *offset* field indicates the position of another field *payload* in the input raw data. Similar to the *length* type field, the related system API semantics imply the *offset* field and its related *payload* variable. Use the following APIs as an example. We can identify the corresponding (second) variable as the *offset* type if the variable is tainted. Based on this, we can further recognize the tainted buffer accessed after the API as the *payload* field.

```
fseek(stream, offset, whence)
lseek(file, offset, whence)
```

• *D[payload, category]*: The *category* field determines the type of another field *payload*. Generally, when a different category value is set up or changed, the structure of the corresponding payload is totally different. The payload-parsing process dramatically varies. For example, as shown in our motivating example, the type field $box \rightarrow type$ determines the parsing logic against the buffer $box \rightarrow data$. Thus, when fuzzing the target program, the field category and its payload structure should be thoroughly understood and identified. As the following code shows, the *category* field often appears in condition statements, e.g., if-else and switch-case.

```
if (category == type1) {
    e = buffer[8] + buffer[12]
    ...
} else if (category == type2) {
    ...
```

Therefore, we design the following strategies to identify the *category* and *payload* fields. First, if a tainted variable appears in conditional checks, for example, in the compare and switch instructions, the variable may be of a *category* field. Please note that in the two operands of the conditional instruction, only one operand should be tainted. This means, the other operand should be a constant or untainted value. Therefore, for such a case, we can group related operands (e.g., type1 and type2) of the conditional checks at the same layer (the if-else or switch-case structure) as the value-selection set for such a *category* field. Second, for each branch body, we collect and group tainted variables. Similar to how we deal with the loop body, these tainted variables are saved in a temporary list, which will be dealt with in the next stage to construct the corresponding *payload*.

4.2.3 Recognizing Hierarchy Dependencies. As presented in Section 2, multiple fields in the input can be grouped into a substructure, while substructures and fields can be further combined to form a larger substructure. The core observation for identifying substructures is that programs often use recursive logic, such as loops and recursive function calls, to process nested inputs. Thus, we can split the whole input into small pieces by analyzing this recursive logic. However, this is a challenging task as this logic often be re-used to handle similar input structures with different contexts during execution. For example, the substructures in the *moof* and *traf* structure are all processed in the same function *box_array_read* at Line 17 in Figure 2. Thus, to precisely identify the substructures, we should record every execution instance of this recursive logic.

Algorithm 2 Build *Input Processing Tree*

```
1:  struct Node {type; start; end; child}
2:  set api_set = {func1, func2, ...}                    ▷ Data access related APIs
3:  global IPT = NewNode(Func, 0, 0, ∅)                  ▷ Input Processing Tree
4:  global cur_node = IPT
5:  function AddInnerNode(type)                          ▷ Add inner node
6:      node = NewNode(type, 0, 0, ∅)
7:      cur_node.addChild(node)
8:      cur_node = node
9:  function AddLeafNode(operand)                        ▷ Add leaf node
10:     tag = getTaintTag(operand)
11:     if tag then
12:         node = NewNode(Load, tag.start, tag.end, ∅)
13:         cur_node.addChild(node)
14: function RollBackNode
15:     node = cur_node.parent
16:     CalculateNodeBound(cur_node)                     ▷ Update the start and end property
17:     if cur_node.start == 0 and cur_node.end == 0 then
18:         DeleteNode(cur_node)
19:     cur_node = node
20: function Instrument(program)
21:     for each BB in each Func do
22:         for each Inst in BB do
23:             if IsCallInst(Inst) then
24:                 Insert(Inst, AddInnerNode(Call))
25:                 if getCallee(Inst) in api_set then
26:                     log(Inst)
27:                 Insert(NextInst(Inst), RollBackNode())
28:             else if IsLoopInst(Inst) then
29:                 Insert(Inst, AddInnerNode(Loop))
30:                 for each Iter in Inst do
31:                     Insert(Inst, AddInnerNode(Iter))
32:                 for each BB in getExitBlocks(Inst) do
33:                     Insert(getFirstInsertion(BB), RollBackNode())
34:             else if IsSwitchInst(Inst) then
35:                 Insert(Inst, AddInnerNode(Switch))
36:             else if IsCmpInst(Inst) then
37:                 Insert(Inst, AddInnerNode(Cmp))
38:             else if IsLoadInst(Inst) then
39:                 Insert(Inst, AddLeafNode(getOperand(Inst)))
```

To mitigate this, we design a novel data structure, namely *Input Processing Tree*, that can help to record every operation launched by the program on processing the input. Thus, it can identify different substructures by recording the executions of recursive logic as well as representing the input processing logic in the target program. We define the crucial data structure *Input Processing Tree* as follows.

Definition 4. An *Input Processing Tree* is an ordered and rooted tree. It consists of the instructions corresponding to the input processing logic and their execution context. It has the following properties.

(1) An inner node is an instruction leading to other instructions. It usually has three types: function prototype, branch, and loop instructions. The sub-tree led by such an internal node must involve tainted data.
(2) A leaf node is an input-accessing instruction whose operands are tainted.
(3) A node is the child of another node (i.e., function/loop/branch body) if the instruction is included in the corresponding internal node.
(4) Any sub-tree contains a node involving tainted variables.

Based on the above definition, we design a novel algorithm (Algorithm 2) to construct the *Input Processing Tree*. In Algorithm 2, we compute the start and end properties (i.e., the boundaries) for each internal node in the *Input Processing Tree*. For nodes of type Function, Loop, and Iteration, we consider their corresponding input as a substructure, and their child nodes correspond to the fields or substructures within the substructure.

In general, *Input Processing Tree* is generated and built online during our taint analysis. An *Input Processing Tree*'s root node is usually a 'main' function node. Then, each instruction is carefully dealt with (Lines 20 to 39 of Algorithm 2). When encountering a function call, we generate a new child node of the function type by inserting a function call to *AddInnerNode* (Line 24). If the instruction's callee is one of the APIs related to data access, we record the inter-field dependency it represents (Line 26). After returning from the function, we insert an instruction to call the *RollBackNode* function (Line 27). This calculates the start and end properties of the function node (Line 16) by traversing its children. Its start property is the minimum value of all its children's start properties, and its end is the maximum end value of all its children. If the sub-tree led by this function node does not have any leaf nodes that have taint tags, resulting in both its start and end being 0, it will be removed.

For a loop block, we create a new child node of the loop type and place it as a child of the corresponding function node being analyzed at Line 29. Since the loop structure will be recursively executed and parsed, we create a new node to represent each iteration and add it under the loop node as a child node at Line 31. Similarly, we insert an instruction to call the *RollBackNode* function (Line 33) after each loop iteration exits.

For a cmp or a switch instruction to be analyzed, we create a unique child node and add it as the child of the function or loop being executed (Lines 35 and 37). For an input-accessing input instruction (e.g., load instruction), we create a new child node by inserting a call instruction to the function *AddLeafNode*. This function extracts the taint information of the operator of the instruction (Line 10) and creates a new leaf node at Line 12. Notice that, all the leaf nodes are input-accessing instructions and any inner node in *Input Processing Tree* should lead a sub-tree whose leaf nodes are input-accessing instructions that have tainted operands. Otherwise, it will be removed (Line 18).

When constructing the input structure, NestFuzz prunes useless sub-trees and merges redundant nodes. Technically, there are two main cases. First, a program may access and load the same variable or different variables but with the same taint sources many times in practice. Consequently, the originally generated *Input Processing Tree* contains many nodes that have the same start and end properties. If these nodes have the same parent nodes, *Input Processing Tree* only needs to keep one of them. Second, the program may access the bytes in one indivisible field separately, thus making *Input Processing Tree* include many redundant nodes. For example, the program may access a four-byte length size field byte by byte. NestFuzz will check and merge them.

Running Example. To clarify the construction of our *Input Processing Tree*, we utilize our NestFuzz approach on the motivating example (in Figure 2) and present its outcome in Figure 5. The leaf nodes in Figure 5 are the load instructions containing the tainted variables, i.e., at Line 12 and 20 in Figure 2. These instructions are the same instructions but executed multiple times under different contexts. Their operands have different taint tags and thus we create distinct nodes to represent them individually. It is worth noting that Lines 12 and 13 both load the variable *type* and we eliminate the duplication. The inner nodes in Figure 5 correspond to function calls, loops, and loop iterations as shown in Figure 2, and the leaf nodes of the subtree led by each internal node are all input-accessing instructions. Note that some program functions and iterations can be executed multiple times, e.g., the function *box_parse_ex*. We create different nodes to represent each execution round.

Interestingly, the structure of the program's input processing tree is similar to the nested structure of the input. In other words, a certain substructure in the input is usually processed in a sub-tree led by a certain function or loop. This relationship can help us identify the input structure.

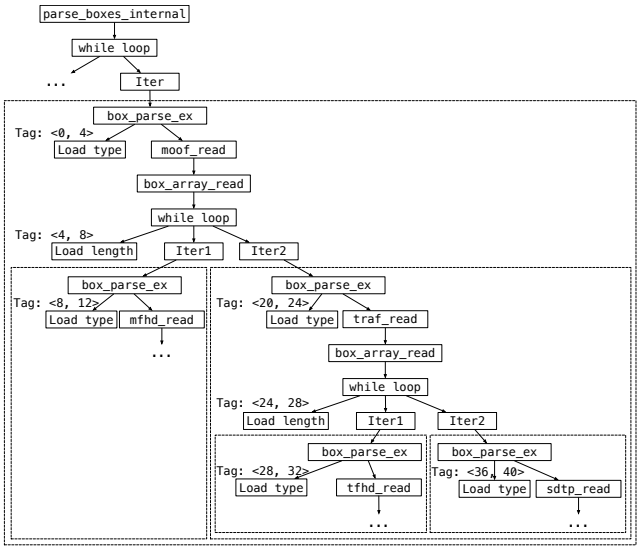


Figure 5: The *Input Processing Tree* for motivating example.

4.3 Fuzzing with Dependency-aware Mutation

After modeling the input processing logic of the program, NestFuzz guides fuzzing with a cascading dependency-aware mutation. That is, when mutating a field or substructure of the input, the mutation operation propagates along the *Input Processing Tree* and cascadingly mutates the related fields and substructures to fix the corresponding dependencies.

4.3.1 Dependency-aware mutation. Considering the inter-field dependency and nested input structure, we design a dependency-aware mutation strategy. As *Input Processing Tree* demonstrates, a simple mutation of an input field changes not only its own properties but also its container (parent) fields and the fields that depend on it. In general, if the fuzzer mutates fields without restoring the broken input structure that the mutation causes, there is a high probability that the generated input will be rejected by the program and cannot explore the deep code space. To resolve this issue, we

propose our cascading mutation strategy, whenever NestFuzz mutates (field or structure-level) the input, it cascadingly mutates other affected fields or substructures to maintain the structure validity of the generated test case.

Specifically, the mutation strategy of NestFuzz can be divided into two levels: field-level and structure-level:

- **Field-level mutators.** NestFuzz designs field-level mutators based on the inter-field dependency knowledge, i.e., *D[payload, length]*, *D[payload, offset]*, and *D[payload, type]*. For instance, for the inter-field dependency *D[payload, length]*, NestFuzz may increase the value of the *length* field by X and insert X bytes after the *payload* correspondingly, or insert a substructure after the payload and increase the value of the *length* field by the length of the substructure.
- **Structure-level mutators.** NestFuzz implements four types of structure-level mutators based on the knowledge of the nested hierarchy, i.e., substructure insertion, deletion, exchange, and file splicing. For instance, to explore more complex substructures, NestFuzz may copy a substructure and insert it into the payload of another substructure.

Based on these mutation operators, our cascading mutation can be described as follows.

In the path exploration stage, when NestFuzz uses any of the mutators, it will search for the affected fields or substructures according to inter-field dependencies and hierarchy dependencies, and apply the corresponding mutator to maintain the dependency relationship from being broken. For instance, suppose NestFuzz inserts payloads to *smp_info* of a substructure (*sdtp*) of Figure 1. In that case, since the length of *sdtp* has changed, NestFuzz will change the value of its *len* field. Moreover, since *sdtp* serves as the payload for a higher-level substructure (*traf*), which means that the *len* field of *traf* will also be adjusted accordingly.

In the security exploitation stage, in order to exploit deep vulnerabilities hidden in the program, NestFuzz prioritizes mutating newly discovered and deeply nested substructures. As shown in our *Input Processing Tree*, nested structures at deeper levels often correspond to the deeper processing logic. On the one hand, this part of the code is explored less frequently, making it more likely to contain vulnerabilities. On the other hand, by mutating these nested structures while maintaining the dependencies between them and the upper-level structures, we can explore the deeper code while still preserving the shallow-level processing logic. Note that, in order to cover bugs in input processing logic, NestFuzz would not *strictly* fix all the dependencies. For example, NestFuzz may only mutate the *length* field (e.g., set its value to the length of the file) but do not adjust the *payload* field (e.g., insert X bytes into the *payload*).

Table 2 shows some examples of the mutators that NestFuzz implements. In total, NestFuzz implemented over 50 mutators at both field and structure levels. Due to the space limitation, we depict a selection of representative examples in the table.

Compared to our multi-layer mutators, AFLSmart introduced three structural mutators. However, its mutators may disrupt inter-field and hierarchy dependencies, resulting in invalid inputs. For instance, it may add a new structure to a *payload* without appropriately mutating the corresponding *length* field. In contrast, our NestFuzz mutators are dependency-aware and thus can mutate multiple fields together to maintain their dependencies, e.g. *length* and *payload*. Furthermore, NestFuzz can be used cascadingly to generate complex and valid inputs (Table 2).

Table 2: Exmaple of mutators implemented by NestFuzz.

		Exploration Mutators	Exploitation Mutators
Field Level	D[payload, length]	length is increased by X →Insert X random bytes after payload	Only increase length by X
		SF is inserted after payload →Increase length by len(SF)	Set length to len(input)
		SF is inserted before payload →Increase length by len(SF)	Delete X bytes from payload
	D[payload, offset]	SF is inserted before payload →Increase offset by len(SF)	Only increase offset by X
		SF is inserted after offset.parent →Increase offset by len(SF)	Set offset to len(input)
	D[payload, category]	Set category to one of the values in the val_set	Bitflip
Structure Level	Insertion	SF1.parent == SF2.parent → copy SF1 and insert after SF2	copy SF1 and insert after SF2
		copy SF1 and insert into the payload of SF2	copy SF and insert at any position
	Deletion	-	delete SF
	Exchange	SF1.parent == SF2.parent → exchange SF1 and SF2	exchange SF1 and SF2
	Splice	SF1 ∈ Input1, SF2 ∈ Input2 SF1.parent == SF2.parent → copy SF1 and insert after SF2	SF1 ∈ Input1, SF2 ∈ Input2 copy SF1 and insert after SF2

5 EVALUATION

5.1 Experiment Setup

Prototype Implementation. We implement NestFuzz with over 3.5k lines of C/C++ code and 1.8k lines of Rust code. Our taint analysis module is implemented based on DFSan [5]. For building *Input Processing Tree*, we use LLVM Pass [22] to instrument the program in several instructions (e.g., load, loop, call, cmp, and switch). We write our runtime building logic in Rust, and also model crucial functions (e.g., *fread*, *memcpy*, *memmove*, *strncpy*, *read*, *pread*, *lseek* and *fseek*). For structure-aware fuzzing, we implement our novel dependency-aware mutation strategies in AFL 2.57b, and develop a python script (0.5k LoC), to synchronize seeds and structure files between fuzzer and input structure modeling.

Benchmark Programs. We selected our benchmark programs from two widely used fuzzing benchmarks, namely FuzzBench [29] and UniFuzz[24], and the testing targets of other structure-aware fuzzers [17, 19, 25, 28]. The main program selection criterion and goal is to cover more programs with different file formats (as input). As a result, our benchmark consists of 20 popular heavily-tested programs, including 14 binary programs (featuring 12 distinct file formats, e.g., image, audio, video, and network packet) and 6 non-binary programs with unique formats (e.g., JSON, and XML). More details are shown in Table 3.

Baseline Fuzzers. In our comparison experiments, we select eight state-of-the-art open-source fuzzers as our evaluation baseline, including universal and frequently-used fuzzers: AFL (2.57b) [30], AFLFast [10], AFL++ [18], and MOpt [28], and input structure-aware fuzzers: AFLSmart [31], WEIZZ [17], ProFuzzer [43], and TIFF [20]. AFL was one of the most popular fuzzer, while AFLFast was built based on AFL by optimizing power scheduling. AFL++ was a fuzzing framework that incorporated several other fuzzing

researches and was one of the highest-rated fuzzing tools according to FuzzBench [29]. MOpt was also based on AFL by optimizing mutator scheduling. AFLSmart, WEIZZ, TIFF, and ProFuzzer leveraged input-format templates or inferences to increase fuzzing efficiency.

Test Bed and Initialization. We carry out all the following experiments on a Ubuntu 20.04 server with AMD EPYC 7513 CUPs (128 cores in total) and 1024 GB RAM. For initialization, we randomly select one initial corpus for each program in our benchmark, and use the same seeds for all fuzzers. The fuzzer was executed on each target application under identical configurations, with each fuzzer bound to a single CPU core. Because ProFuzzer runs in two separate processes, one for fuzzing and one for probing, and our current implementation for NestFuzz also runs in two separate processes, one for fuzzing and one for input processing logic modeling, we assign two CPU cores to both ProFuzzer and NestFuzz.

Table 3: The benchmark programs and parameters used in our evaluation.

Format	Library	Version	Program & Parameter
ZIP	p7zip	16.02	7za t @@
JPG	exiv2	0.26	exiv2 @@
MP4	gpac	2.0.0	MP4Box -diso @@
GIF	imageMagick	7.1.0-49	magick identify @@
H.265	libde265	1.0.9	dec265 @@
MOBI	libmobi	0.11	mobitool -e -o /tmp @@
PNG	libpng	1.6.38	pngtest @@
MP3	mp3gain	1.5.2	mp3gain @@
MP4	bento4	1.5.1-628	mp42aac @@ /dev/null
ELF	binutils	2.28	nm-new -A -a -l -S -s –special-syms –synthetic –with-symbol-versions -D @@
ELF	binutils	2.28	objdump -S @@
PDF	xpdf	4.00	pdftotext @@ /dev/null
PCAP	tcpdump	4.8.1 (libpcap 1.8.1)	tcpdump -e -vv -nr @@
TIFF	libtiff	3.9.7	tiffsplit @@
TXT	ncurses	6.1	tic -o /dev/null @@
MD	cmark	0.30.3	cmark @@
YAML	libyaml	0.2.5	run-parser @@
XML	libxml2	2.11.0	xmllint –testIO @@
JSON	jansson	2.14	jansson @@
C	indent	2.2.13	indent @@ -o /dev/null

5.2 Test Coverage

We run all fuzzing tools with the same timeout of 24 hours and repeat the experiments five times for statistical power. Table 4 reveals the line coverage and branch coverage of each fuzzing tool. Our evaluation results prove that NestFuzz outperforms the state-of-the-art format-aware and other fuzzers in line and branch coverage.

For binary file formats, NestFuzz obtains the highest number of lines and branches in 13 out of the 14 benchmark programs. On average, NestFuzz achieves 27%/56%/28%/154% more line coverage and 28%/67%/30%/204% more branch coverage than AFLSmart/WEIZZ/ProFuzzer/TIFF, respectively, compared with structure-unware but general fuzzers, NestFuzz achieves 40%/48%/30%/32% more line coverage and 47%/57%/31%/35% more branch coverage that AFL/AFLFast/AFL++/MOpt, respectively. Although our current implementation does not consider the checksum used by the PNG format to check file integrity, NestFuzz still achieves the second-best coverage results during testing of *pngtest*.

For non-binary file formats, in comparison to the state-of-the-art structure-aware fuzzing approaches, we find NestFuzz can achieve the best results in most cases. For instance, NestFuzz averagely has 10.89%/49.01% more line coverage and 16.51%/73.38% more branch coverage than WEIZZ and TIFF. In contrast to the general fuzzers, the experiments show NestFuzz's results are better than AFL and AFLFast, and slightly below AFL++ and MOpt. One important reason is that many non-binary formats, e.g., JSON and C language files, are *grammar-based* and characterized by strict syntactic constraints. Slightly differing in perspective, our main focus mainly centers on *chunk-based* inputs [17] that have complex hierarchical structures.

To further demonstrate the effectiveness of NestFuzz, we track the growth trend of edge coverage for AFL-based fuzzers and present it in Figure 6. In most of the projects, NestFuzz achieves the highest edge coverage at a faster pace, compared to all other fuzzers. NestFuzz achieves significant improvements distinctly in the two following types of programs.

First, NestFuzz outperforms the baseline fuzzers distinctly when testing *complex* file formats. Take the MP4 file format as an example. When testing the MP4 program *MP42aac* [2], NestFuzz achieves 104% more line coverage and 115% more branch coverage than AFLSmart, which achieves the best result in the baseline fuzzers. We further analyze the result and find the reasons mainly contain two parts. First, template-based fuzzers like AFLSmart need a precise format model for input format, which is challenging to write for complex file formats like MP4. Actually, the specification of MP4 defines 104 different and nested data boxes. It is difficult to write a model that covers all the data boxes and clearly outlines all the potential associations they might have. As a result, the MP4 format model supplied by AFLSmart only contains 7 different boxes (6.7% of the specification). The second reason is that, there are inconsistencies between format specification and program implementation.

Second, NestFuzz outperforms the baseline fuzzers when testing programs that can process *multiple* file formats. For instance, *magick* [4] can edit more than 200 image formats, and NestFuzz achieves 59% more line coverage and 66% more branch coverage than AFL when testing it. After manually checking the seeds generated by these fuzzers, we find that the main reason is, benefiting from our structural-level mutators, NestFuzz could mutate the seeds with a big change at the substructure level and keep the newly generated seeds valid. Thus, it has more chances to take a big step in the search space of input formats and find a new format.

5.3 Vulnerability Detection

5.3.1 Unique bugs. In order to study whether the coverage improvements NestFuzz achieves can help it trigger more vulnerabilities hidden in the program, we compare the vulnerability discovery ability of NestFuzz with other fuzzers.

We compile the programs in our benchmark (Table 3) with ASAN [34] and fuzz them for 24 hours and repeat the experiment five times. To categorize the program crashes into distinct bugs, we follow the procedure proposed in previous studies [24], involving the following steps. First, we used AFL-cmin [30] to minimize the

Table 4: Average line and branch coverage(for 24 hours over five runs) achieved by fuzzers when testing real-world programs.

Input Format	Program	Structure-aware Fuzzers										General Fuzzers							
		NestFuzz		AFLSmart		WEIZZ		ProFuzzer		TIFF		AFL		AFLFast		AFL++		MOpt	
		L	B	L	B	L	B	L	B	L	B	L	B	L	B	L	B	L	B
Binary Formats	7za	**9762**	**7554**	8501	6637	7865	5824	8245	6435	5004	3702	7816	6006	7724	5950	7408	5538	8219	6208
	exiv2	**5659**	**6722**	4454	4944	3798	4134	4308	4797	1160	1151	3675	3971	2358	2507	4249	4821	4288	4810
	MP4Box	**9395**	**5677**	6335	3632	4809	2701	6280	3683	3696	2074	4648	2713	5035	2876	6270	3654	5669	3333
	magick	**22655**	**11354**	14063	6815	8688	3712	14148	6850	6129	2710	14274	6839	13146	6115	13049	6507	12557	6160
	dec265	**10660**	**5790**	10645	5754	10199	5036	10649	5765	9761	4014	10437	5404	10561	5598	10668	5766	10732	5777
	mobitool	**3543**	**2267**	–	–	3106	1988	3025	1941	2865	1689	2976	1873	2971	1867	3013	1931	3029	1941
	pngtest	2371	1232	1979	975	**2523**	**1343**	1971	969	1919	923	1938	944	1944	947	1979	975	1983	979
	mp3gain	**2085**	**1011**	1930	925	2037	1001	1960	934	1632	762	1948	947	1936	931	1958	954	1973	963
	mp42aac	**5242**	**3654**	2567	1700	2114	1299	2516	1651	1895	1146	2304	1443	2295	1449	2366	1559	2377	1545
	nm-new	**4204**	**2444**	3391	1968	3351	1836	3410	1981	1264	610	3339	1947	2972	1756	3111	1904	3401	1967
	objdump	**7450**	**4740**	6363	4032	4177	2441	6221	3962	2000	998	6238	3894	5191	3268	5985	3909	6171	3917
	pdftotext	**3191**	**1914**	2638	1455	1880	1060	2873	1591	1427	763	1845	1046	1843	1045	2895	1615	1848	1051
	tcpdump	**19431**	**13549**	17562	12263	12298	7651	16608	11509	2307	1359	12950	8543	12721	8501	18093	12398	16975	11407
	tiffsplit	**3219**	**1957**	–	–	3052	1834	2988	1811	1878	1031	3051	1828	2825	1682	2971	1810	3004	1813
	Average	**7776**	**4990**	6702	4258	4993	2990	6069	3837	3067	1638	5531	3386	5252	3178	6001	3810	5873	3705
	Increase	–	–	26.95%	28.46%	55.75%	66.89%	27.77%	29.67%	153.55%	204.67%	40.58%	47.40%	48.07%	57.03%	29.58%	30.98%	32.40%	34.69%
Non-binary Formats	tic	1868	1478	–	–	1674	1295	1837	1450	–	–	1544	1166	1609	1229	1872	1478	**1905**	**1514**
	cmark	10504	9301	–	–	8650	7389	10879	9474	4767	3608	11038	9546	9698	8662	**11480**	**10019**	11349	9848
	run-parser	**1636**	**2340**	–	–	1529	1852	1547	2210	940	925	1544	2122	1540	2104	1544	2219	1547	2218
	xmllint	9021	7646	–	–	8452	6888	8741	7319	7741	5876	8141	6608	8151	6609	**9137**	**7823**	8697	7274
	jansson	**925**	**439**	–	–	890	427	911	441	725	411	891	432	894	433	896	435	903	439
	indent	2298	1651	–	–	**2480**	**1765**	2326	1672	2190	1509	2288	1636	2295	1645	2360	1694	2328	1670
	Average	4375	3809	–	–	3946	3270	4374	3761	3273	2466	4241	3585	4031	3447	**4548**	**3945**	4455	3827
	Increase	–	–	–	–	10.89%	16.51%	0.04%	1.28%	49.01%	73.38%	3.17%	6.25%	8.55%	10.50%	-3.80%	-3.43%	-1.78%	-0.47%

[1] The L means line coverage, the B means branch coverage, and – means the program cannot be fuzzed by the fuzzer.

set of inputs that resulted in program crashes. This involved identifying the smallest possible subset of files in the input directory that could still trigger the full range of instrumentation data points observed in the initial corpus. Then, we relied on the output report generated by ASAN to extract the top three functions in the stack trace and group them as a triple to detect and eliminate duplicate bugs. Bugs were deemed unique if they had different triples and vulnerability types. Finally, we manually analyzed the results and counted the corresponding number of unique bugs.

Table 5 presents the average and maximum number of unique bugs of several fuzzing tools over five runs. NestFuzz finds the largest number of unique bugs in most of the programs. In detail, NestFuzz on average found 247 unique bugs, which is 578.57% more than the baseline fuzzer, i.e., AFL, and 107.56% more than the second-best fuzzer AFLSmart. In the maximum run, NestFuzz found 296 unique bugs, which is 572.73% more than the baseline fuzzer AFL, and 75.15% more than the second-best fuzzer AFLSmart.

To further prove the effectiveness of NestFuzz, we plot the growth trend of unique bugs in Figure 7. Our results show that NestFuzz outperforms all other fuzzers by detecting a larger number of unique bugs at a faster rate in these projects.

5.3.2 Exposing Known Vulnerabilities. To better demonstrate NestFuzz's ability to discover vulnerabilities, we match the unique bugs discovered by the fuzzing tool with known vulnerabilities (already assigned CVE IDs). To achieve this goal, we used scripts provided by UniFuzz [24] and manually confirmed the matching results. Table 6 shows the shortest time taken by different fuzzing tools to detect certain known vulnerabilities across five runs. In particular, within 24 hours, NestFuzz was able to detect all 20 known vulnerabilities, whereas AFLSmart, WEIZZ, ProFuzzer, AFL, AFLFast, AFL++, and MOpt only discovered 10, 12, 10, 9, 9, 8 and 9 respectively. When testing the program *mp42aac*, only NestFuzz identified unique bugs (as presented in Table 5) as well as known vulnerabilities. It took a mere 0.55 hours for NestFuzz to uncover vulnerability CVE-2018-14531; other fuzzers were unable to do so even after 24 hours of testing.

5.3.3 Zero-day Vulnerability Discovery. Table 5 also aggregates the unique vulnerabilities only discovered by NestFuzz on the programs in our benchmark. Totally, NestFuzz found 46 unknown vulnerabilities and we reported them to corresponding vendors. Currently, 39 have been confirmed and 37 of them have been assigned with CVE-ids until the writing of this paper. To demonstrate how the structure knowledge NestFuzz learned helps it hack the program, we use a zero-day vulnerability it found for a case study.

CVE-2022-48065: Memory Leak. NestFuzz discovered a memory leak vulnerability in *nm-new*, which is part of Binutils [6]. The vulnerable code is displayed in Figure 8. In line 3429, the function parses data in the DWARF format `.debug_info` section of the ELF file using a loop (line 3594) to parse its abbreviations. The bug occurs when a nested abbreviation contains both a `DW_AT_decl_file` type of data box and a `DW_AT_specification` type of data box. Furthermore, the `DW_AT_specification` box has a child box that contains another `DW_AT_decl_file`. When processing the outermost `DW_AT_decl_file`, memory is allocated at line 3633, and then recursively calls the function `find_abstract_instance` to parse

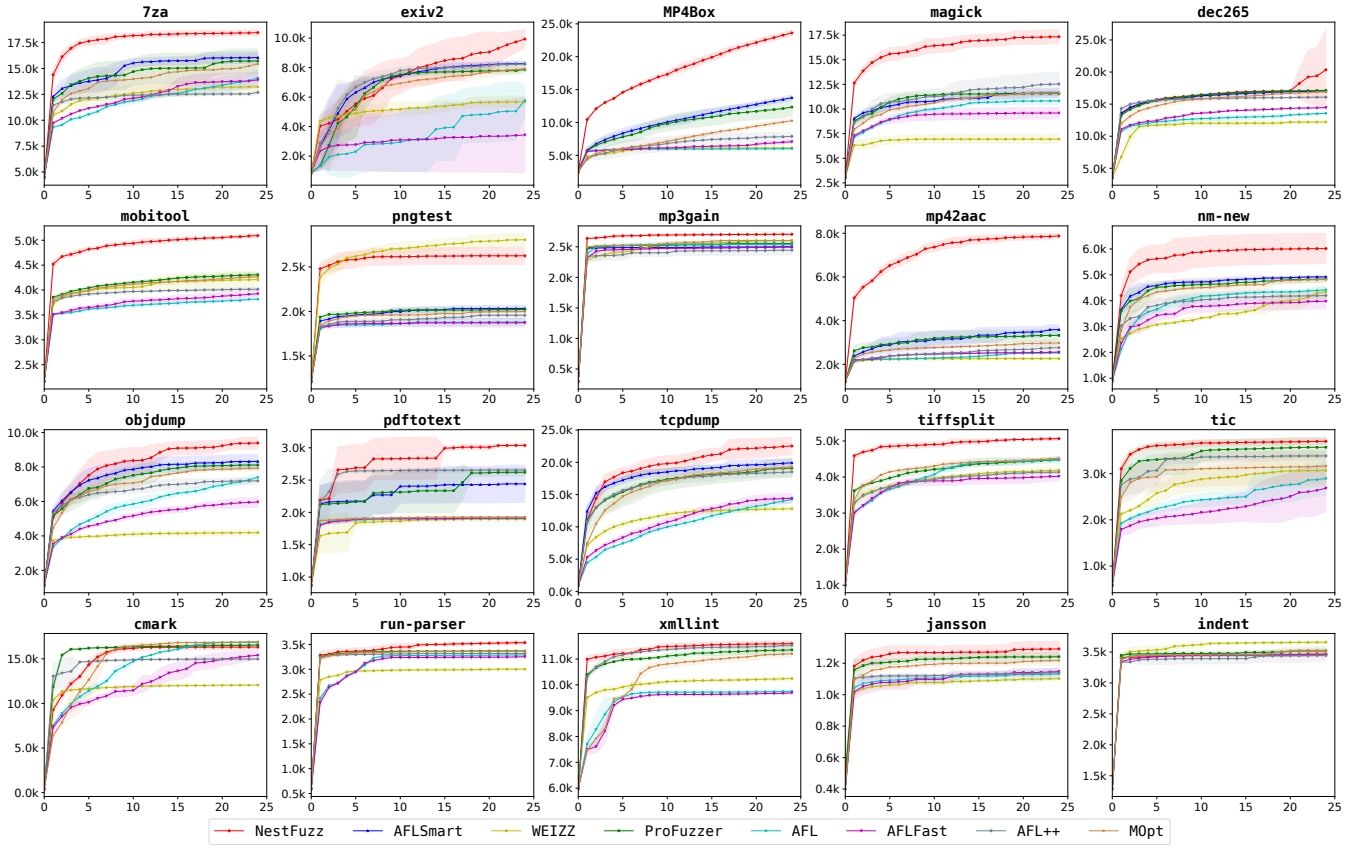


Figure 6: The arithmetic mean edge coverage of AFL-based fuzzers running for 24 hours and one standard deviation error bars over five runs.

Table 5: Average and maximum number of unique bugs(for 24 hours over five runs) achieved by various fuzzers when testing real-world programs.

Program	NestFuzz		AFLSmart		WEIZZ		ProFuzzer		AFL		AFLFast		AFLplusplus		MOPT		Bugs found only by NestFuzz		
	Average	Max	Average	Max	Average	Max	Average	Max	Average	Max	Average	Max	Average	Max	Average	Max	Unknown	Known	CVE
7za	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1
exiv2	13	18	2	3	4	5	2	4	1	2	0	0	4	5	2	3	0	20	0
MP4Box	3	8	0	0	0	0	0	0	0	0	0	0	0	0	0	0	9	0	8
dec265	30	46	23	49	0	0	9	18	0	0	0	1	26	43	4	9	19	1	16
mp3gain	6	6	6	7	6	9	7	8	4	4	4	5	5	5	6	6	0	1	0
mp42aac	13	14	0	0	1	1	0	0	0	0	0	0	1	1	0	0	12	2	8
nm-new	3	5	0	2	3	5	2	4	0	0	0	0	0	1	0	1	2	4	2
objdump	10	11	2	3	2	3	2	3	0	1	1	1	2	3	1	3	2	12	1
pdftotext	1	1	0	1	0	0	0	1	0	0	0	0	0	1	0	0	0	0	0
tcpdump	149	165	86	104	23	30	76	83	23	28	28	41	18	24	36	50	0	129	0
tiffsplit	13	15	0	0	23	24	7	10	7	8	8	8	9	11	7	10	0	2	0
tic	4	5	0	0	4	6	5	8	0	0	1	2	3	6	5	7	1	1	1
indent	1	1	0	0	1	1	2	3	1	1	1	1	3	6	4	7	0	0	0
Total	247	296	119	169	67	84	112	142	36	44	43	59	71	106	65	96	46	172	37
Increase	–	–	107.56%	75.15%	268.66%	252.38%	120.54%	108.45%	578.57%	572.73%	474.42%	401.69%	247.89%	179.25%	280.00%	208.33%	–	–	–

[1] Since WEIZZ (implemented based on QEMU [9]) and TIFF (implemented based on Pintool [27]) were incompatible with ASAN [34], we initially tested them on non-ASAN-instrumented programs and then replayed the generated inputs on ASAN-instrumented programs.

[2] TIFF fuzzer did not find any bugs in all the programs in our evaluation.

[3] All fuzzers, when testing programs *magick*, *mobitool*, *pdftotext*, *cmark*, *run-parser*, *xmllint*, and *jassson*, did not discover any bugs.

the innermost child's `DW_AT_specification`. Lines 3629-3635 process this inner child's `DW_AT_decl_file`, reallocating memory for pointer variable *filename_ptr* before freeing it, resulting in memory leakage. Traditional byte- or field-level mutation strategies are unlikely to generate such nested structures for this vulnerability.

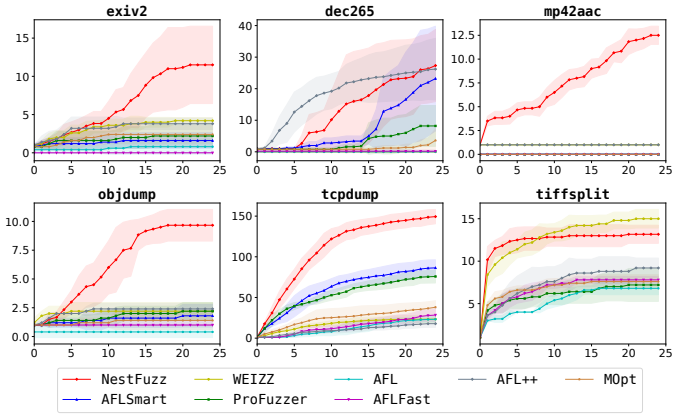


Figure 7: The arithmetic mean number of unique bugs of AFL-based fuzzers running for 24 hours and one standard deviation error bars over five runs.

Table 6: Time consumption to expose known vulnerabilities.

Program	CVE ID	Exposing Time (hours)							
		NestFuzz	AFLSmart	WEIZZ	ProFuzzer	AFL	AFLFast	AFL++	MOpt
exiv2	CVE-2017-11339	16.25	20.72	17.92	4.5	T/O	T/O	11.52	3.42
	CVE-2017-17669	10.63	T/O	T/O	T/O	T/O	T/O	T/O	T/O
	CVE-2018-10780	16.2	21.38	19.3	T/O	T/O	T/O	11.77	T/O
tcpdump	CVE-2016-7973	2.57	6.39	17.45	2.29	19.17	22.82	T/O	9.82
	CVE-2016-7985	1.02	T/O	18.32	T/O	18.17	22.19	T/O	T/O
	CVE-2016-7983	0.62	4.52	13.87	T/O	11.44	11.66	T/O	T/O
	CVE-2017-13013	3.78	7.42	T/O	8.13	T/O	T/O	T/O	T/O
	CVE-2017-12986	13.75	T/O	T/O	T/O	T/O	T/O	T/O	T/O
	CVE-2017-13013	3.78	7.42	T/O	8.13	T/O	T/O	T/O	T/O
mp3gain	CVE-2017-14409	0.03	0.13	0.21	0.14	0.28	0.33	0.24	0.32
	CVE-2017-14410	0.28	0.12	0.67	0.2	1.13	0.87	0.25	0.65
	CVE-2017-14406	1.83	22.32	3.33	T/O	6.81	4.58	1.4	1.19
	CVE-2017-14407	0.11	1.27	2.98	0.28	9.43	4.92	4.28	1.24
mp42aac	CVE-2018-14587	16.21	T/O	T/O	T/O	T/O	T/O	T/O	T/O
	CVE-2018-14584	4.4	T/O	T/O	T/O	T/O	T/O	T/O	T/O
	CVE-2018-14531	0.55	T/O	T/O	T/O	T/O	T/O	T/O	T/O
	CVE-2018-14588	20.35	T/O	T/O	T/O	T/O	T/O	T/O	T/O
tiffsplit	CVE-2016-9273	0.61	–	0.75	0.53	21.25	12.54	4.56	0.45
	CVE-2015-7554	1.92	–	0.98	17.31	T/O	T/O	0.34	1.63
	CVE-2010-2631	0.02	–	0.01	0.22	0.09	0.01	T/O	2

```
/* File: bfd/dwarf2.c */
3429  static bool find_abstract_instance (...) {
3594      for (i = 0; i < abbrev->num_attrs; i++) {
3600          switch (attr.name) {
3612              case DW_AT_specification:
3614                  find_abstract_instance(...);
3618              break;
                  ...
3629              case DW_AT_decl_file:
3633                  *filename_ptr = concat_filename(...);
3635              break; /* Allocate memory and memory
3642          }             leak happens */
3643      }
3648  }
```

Figure 8: A memory leak vulnerability detected by NestFuzz.

5.4 Ablation Study

NestFuzz learns two types of major knowledge about the input, i.e., the inter-field dependency, and the hierarchy of input structure. We conducted experiments to improve the importance of these two kinds of knowledge for NestFuzz to explore the program. Table 7 presents the line and branch coverage of NestFuzz, NestFuzz-F, which only leverages the inter-field dependency knowledge to mutate the input, and NestFuzz-S, which only leverages the structure's hierarchy knowledge. NestFuzz achieves the highest coverage on all the programs, on average, NestFuzz-S covers 11.51% and 11.40%, and NestFuzz-F covers 24.92% and 26.22% less code and branch coverage than NestFuzz, respectively. We can conclude that both field dependency and structure's hierarchy knowledge are significant for NestFuzz to generate high-quality test cases.

Table 7: Average line and branch coverage(in 3 runs) achieved by NestFuzz, NestFuzz-F, NestFuzz-S when testing real-world programs.

Program	NestFuzz		NestFuzz-F		NestFuzz-S	
	L	B	L	B	L	B
7za	9947	7790	9281(-6.70%)	7092(-8.96%)	8409(-14.65%)	6664(-14.45%)
objdump	7662	4783	6978(-8.93%)	4415(-7.69%)	5741(-25.16%)	3706(-22.52%)
magick	21420	11119	17601(-17.83%)	9427(-15.22%)	1442(-34.44%)	6911(-37.85%)
MP4Box	9369	5656	7938(-15.27%)	4714(-16.65%)	6471(-30.75%)	3801(-32.80%)
MP42aac	5252	3809	4929(-6.15%)	3487(-8.45%)	2748(-47.47%)	1852(-51.38%)
mobitool	3506	2240	3477(-0.83%)	2225(-0.67%)	2922(-14.66%)	1903(-15.04%)
tic	1915	1534	1858(-2.98%)	1481(-3.46%)	1748(-6.53%)	1387(-9.58%)
pdftotext	3189	1899	2191(-31.30%)	1397(-26.43%)	1839(-42.08%)	1049(-44.76%)
tcpdump	18932	13152	16361(-13.58%)	11175(-15.03%)	1711(-8.60%)	12156(-7.57%)
Average Reduction	-	-	11.51%	11.40%	24.92%	26.22%

[1] The L means line coverage, and the B means branch coverage.

5.5 Input Structure Inference

5.5.1 Performance. NestFuzz utilizes dynamic taint analysis and code instrumentation to trace the program and comprehend the input processing logic. However, this approach inevitably introduces additional overhead to the program.

To assess NestFuzz's performance, we conducted a comparative analysis of its time overhead against other input structure inference methodologies, namely WEIZZ, ProFuzzer, TIFF, and AFL-Analyze [1]. AFL-Analyze is a file format analyzer that AFL provides. It identifies the boundaries and types of input fields by sequentially flipping data bytes and observing the behavior of the program. We excluded AFLSmart from our comparison, as it inferred input structures based on templates rather than an automatic program analysis.

Specifically, we collected five inputs that differed widely in size and structure from each program within our benchmark that processes binary input formats. As an illustration, we collected executable files, object code, shared libraries, and core dumps for the ELF format. We then calculated the average time taken by NestFuzz and the other approaches to infer the structure of a single input. The summarized results are presented in Table 8.

Across all programs, NestFuzz demonstrated significantly reduced inference times, averaging about 0.77 seconds. In contrast, the second-best solution, TIFF, required an average of 24.9 seconds. This suggests that the time consumed by NestFuzz for comprehending input structures remains reasonable in practical scenarios.

5.5.2 Complexity and Correctness. NestFuzz adeptly captures inter-field dependencies and hierarchy dependencies within the input by

constructing the *Input Processing Tree*. In this section, we study the complexity and correctness of the input structure that NestFuzz recovers.

Complexity. To investigate the complexity of the tree structure, we conducted two sets of experiments.

On the one hand, we compute the average depth of the constructed tree structure in NestFuzz for each input that we select. Generally, a tree hierarchy with a greater average depth is considered more complex. The outcomes of this analysis are depicted in Figure 9 (a). Specifically, the complexity of the input structure varies across different input formats and programs. While most inputs exhibit an average depth larger than 4, MP4 files display the highest average depth, reaching 11.68 within *MP4Box*. Remarkably, NestFuzz generates distinct tree structures for identical inputs processed by different programs. For instance, when applying the same inputs to both *nm-new* and *objdump*, subtle disparities in the average depth of the constructed tree structure emerge. This phenomenon arises due to potential variations in processing logic among different programs for the same input. In contrast to template-based fuzzing approaches, e.g. AFLSmart, NestFuzz possesses the capability to detect such distinctions. Consequently, NestFuzz can tailor its input mutation strategies to accommodate these differences effectively. Furthermore, the complexity of the input structure also influences the efficacy of various fuzzing methodologies. For instance, as demonstrated in Section 5.2, NestFuzz notably surpasses our baseline fuzzers in testing programs *MP4Box* and *mp42aac*.

Furthermore, we opted for two representative file formats, MP4 as a complex format, and JPG as a simple format as indicated by Figure 9 (a), to study the depth distribution of inputs generated by structure-aware fuzzers during a 24-hour testing period on their processing programs (*MP4Box* for MP4 and *exiv2* for JPG). To begin, we accumulated inputs generated by structure-aware fuzzers during the program testing phase. Subsequently, we utilized NestFuzz to generate a tree structure for each input and subsequently classified the inputs based on the depth of their tree structure. Finally, we determined the proportion of inputs generated by each fuzz testing tool within distinct depth categories. The outcome of our evaluation for the *MP4Box* and *exiv2* programs is illustrated in Figure 9 (b) and Figure 9 (c).

Based on our experimental results, we can draw a conclusion: compared to existing tools, NestFuzz is capable of generating a higher proportion of inputs with relatively complex structures. For instance, only NestFuzz and AFLSmart generated inputs with average structure depths exceeding 40 and 8 when testing *MP4Box* and *exiv2*, respectively. Remarkably, the inputs generated by NestFuzz accounted for 70% and 85% in the respective cases. This unique ability to generate inputs with intricate structures significantly augments NestFuzz's effectiveness in uncovering vulnerabilities that are deeply embedded.

Correctness. In order to study the correctness of the input structure that NestFuzz infers, we selected the MP4 format [3] as a case study. First, drawing inspiration from existing work [35], we utilized a popular tool 010 Editor [39] to parse the five files that we selected and extract their structure information as the ground truth.

Compared to 010 Editor, NestFuzz can even identify more precise input fields and structures. For example, it recognizes 19.66% more fields and 76.48% more structures on average in the MP4 format. We sincerely analyzed these fields and found that they describe the input structure's significant properties and impact the program execution. Figure 10 illustrates the results of identifying an MP4 file using NestFuzz. In Figure 10 (a), the outcomes of field identification are presented, while Figure 10 (b) depicts the recognized structure. In comparison to NestFuzz, the identification carried out by 010 Editor is limited. Specifically, 010 Editor only succeeds in identifying the *length* and *type* fields within the *udta* box. For the *meta* box and its associated sub-boxes, 010 Editor only perceives them as raw data. Upon conducting a thorough analysis, we uncovered that 010 Editor relies on a template to parse the MP4 file. However, this template merely accounts for 37 box types, whereas the actual *MP4Box* program encompasses 532 box types. This discrepancy leads to over 93% of the box types handled by *MP4Box* being unrecognized by 010 Editor. As a consequence, the ability to handle nested structures within inputs, as highlighted in Section 2, is crucial for fuzzers to generate intricate inputs capable of exposing profound vulnerabilities.

Table 8: Average time to process an input(seconds).

Format	Program	Size(bytes)	NestFuzz	WEIZZ	ProFuzzer	TIFF	AFL-Analyze
ZIP	7za	844	0.05	123.47	17792.42	15.50	10.18
JPG	exiv2	3212	0.01	83.61	61234.48	18.51	36.28
MP4	MP4Box	9932	0.06	255.44	T/O	38.82	185.46
MP4	mp42aac	9932	0.05	240.44	T/O	12.38	86.63
GIF	magick	1017	0.04	56.65	5748.73	38.82	12.91
H.265	dec265	3084	4.01	5113.96	T/O	80.51	102.63
MOBI	mobitool	9245	0.27	270.02	T/O	32.67	135.81
PNG	pngtest	1271	0.20	81.35	17208.56	20.13	16.01
MP3	mp3gain	6857	0.01	4636.78	11226.83	11.44	36.10
ELF	objdump	5151	0.07	467.63	T/O	8.93	52.95
ELF	nm-new	5151	0.03	113.68	T/O	5.98	34.50
PDF	pdftotext	9651	5.93	3476.14	T/O	53.06	386.28
PCAP	tcpdump	850	0.01	27.14	9775.93	4.58	4.93
TIFF	tiffsplit	1216	0.02	36.97	66100.30	7.31	5.74

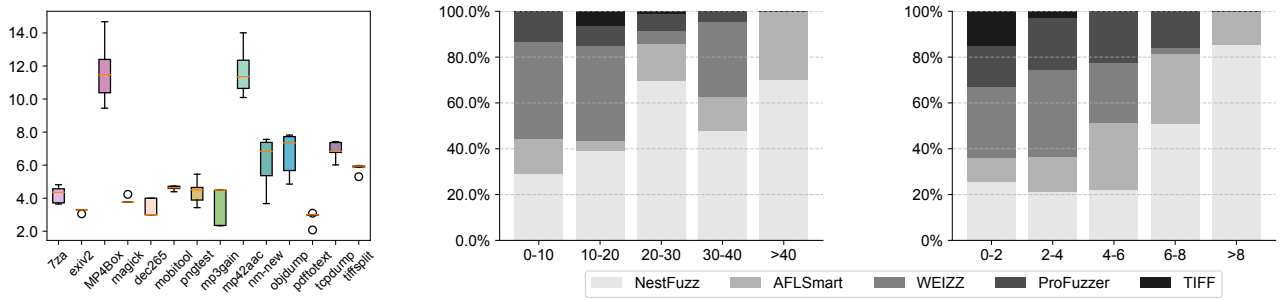
[1] T/O means ProFuzzer fails to get the result within 24 hours limits.

6 DISCUSSION AND LIMITATION

Our experiments show that NestFuzz outperforms the state-of-the-art format-aware fuzzing approaches by modeling the input processing logic of the tested program and gaining a whole understanding of the input structure. Thus, during input mutation, it can utilize a dependency-aware strategy to ensure it would not frequently generate invalid input, which greatly improves the fuzzing efficiency. In this section, we discuss the limitations of NestFuzz.

Limitations due to the dependency on taint tracking. NestFuzz identifies input processing instructions based on byte-level dynamic taint analysis. However, under-taint and over-taint issues may be existing in such a taint analysis, inevitably impacting the precision of NestFuzz. We leave involving the existing mitigation solutions [21] against under-taint and over-taint as our future work.

Generalizing to infer inter-field dependencies of other types. In this work, we mainly focus on three structurally important dependencies. In addition to these, there are still other types of complex constraints over input, e.g. input checksum verification and some

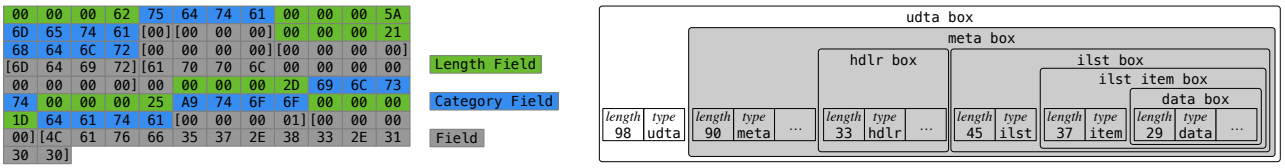


(a) The average depth of the inferred input structures by NestFuzz.

(b) Structure-aware fuzzers' varying input depth ratios for MP4Box.

(c) Structure-aware fuzzers' varying input depth ratios for exiv2.

Figure 9: Input structure complexity comparison.



(a) Fields identified in an MP4 file.

(b) Hierarchy structure of the MP4 file.

Figure 10: The identification results of an MP4 file by NestFuzz.

numerical constraints between fields. For checksum-related constraints, it could be mitigated by instrumenting and patching the checksums [17, 38]. For numerical constraints, it could be dealt with by applying symbolic execution [32, 33, 44] or gradient descent [13, 14]. Furthermore, constraints might exist between certain *string* or *byte-array* fields. Commonly, programs employ functions like *memcmp* to compare two strings or arrays, with these functions frequently utilizing pointer arguments [8, 17]. Therefore, similar techniques used for modeling functions like *read* (See more details in Section 4.2.2) can be extended to infer and handle these constraints. We consider these enhancements as integral parts of our future work.

7 RELATED WORK

Structure-aware Fuzzing The critical idea behind structure-aware fuzzing [8, 17, 20, 31, 35, 42, 43] is to mutate the input more efficiently based on the knowledge about the input structure. For example, TIFF [20] presented mutation strategies for finding memory-corruption bugs by tagging the input bytes with a basic type. ProFuzzer [43] proposed a probing technique to understand the type of the input field, which supported 6 field types. WEIZZ [17] proposed a technique to automatically identify the fields and chunks within chunk-based file formats, inspired by REDQUEEN [8]. AIFORE [35] reversed the input field types and boundaries based on taint analysis and neural network model. Technically, these approaches mainly focused on identifying single fields of input. Compared with existing work, NestFuzz not only identifies single fields but also considers the input's hierarchy structure as well as the complex dependencies across fields and substructures.

Another thread of work utilized pre-defined templates to generate structural input. For example, AFLSmart [31] introduced smart grey-box fuzzing based on Peach [7]. It leveraged a format template to obtain a tree representation of the input structure and implemented smart structure-level mutation strategies. Compared with them, NestFuzz relies on no template and could automatically learn input structures from code implementation.

In general, current format-aware fuzzing tools lacked a comprehensive understanding of input formats. Compared with them, NestFuzz models the input formats with *Input Processing Tree* and proposes a dependency-aware mutation to drive fuzzing, which are proved to be more effective based on our evaluation results.

Input Structure Reverse Engineering. This kind of work [11, 15, 16, 23, 26, 36, 37, 40] focused on automatically reversing protocols, file format structures and specifications. Discoverer [15] and Polyglot [11] focused on identifying fields and separators in protocol messages. However, they assumed the input to be flat and therefore ignored the recognition of nested structure in the input. Tupni [16] reversed the record sequences and types for file formats or protocols, where a record was a contiguous sequence of fields. However, it only concentrated on identifying individual records, disregarding the combination and nesting of records that were prevalent in complex real-world formats such as MP4. Another example is TIE [23], which could reverse data type abstractions (for example, integer) from binary programs. However, these data types were not suitable for guiding fuzzing. For instance, a *length* field might correspond to an *int* variable in the program, but the *int* type was of little effect in guiding the seed mutation.

Different from existing works, NestFuzz learns the knowledge about the inter-field dependencies and hierarchy dependencies by modeling the input processing logic with taint analysis. Our structural knowledge more accurately reflects the program's code implementation, thereby enhancing effective fuzzing.

8 CONCLUSION

In this paper, we propose a smart structure-aware fuzzing approach, NestFuzz. By modeling the input processing logic, it can identify the dependencies across fields and sub-structures, and utilizes a dependency-aware mutation strategy to improve fuzzing efficiency.

Our evaluation results show that NestFuzz substantially outperforms the existing approaches in terms of both code coverage and vulnerability discovery. It discovers 46 zero-day vulnerabilities and 37 have been assigned with CVE-IDs. Our research proves that an in-depth understanding of the input structure is critical for fuzzing to generate high-quality test cases.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their insightful comments that helped improve the quality of the paper. This work was supported in part by the National Key Research and Development Program (2021YFB3101200), National Natural Science Foundation of China (62172104, 62172105, 61972099, 62102093, 62102091). Zhemin Yang was supported in part by the Funding of Ministry of Industry and Information Technology of the People's Republic of China under Grant TC220H079. Yuan Zhang was supported in part by the Shanghai Rising-Star Program under Grant 21QA1400700 and the Shanghai Pilot Program for Basic Research - FuDan University 21TQ1400100 (21TQ012). Min Yang is the corresponding author, and a faculty of Shanghai Institute of Intelligent Electronics & Systems and Engineering Research Center of Cyber Security Auditing and Monitoring.

REFERENCES

[1] 2023. *Automatically inferring file syntax with afl-analyze.* Retrieved 2023-05-05 from https://lcamtuf.blogspot.com/2016/02/say-hello-to-afl-analyze.html
[2] 2023. *Bento4.* Retrieved 2023-05-05 from https://github.com/axiomatic-systems/Bento4
[3] 2023. *ELF Linux manual page.* Retrieved 2023-05-05 from https://man7.org/linux/man-pages/man5/elf.5.html
[4] 2023. *ImageMagick.* Retrieved 2023-05-05 from https://github.com/ImageMagick/ImageMagick
[5] 2023. *LLVM dataFlowSanitizer.* Retrieved 2023-05-05 from https://clang.llvm.org/docs/DataFlowSanitizer.html
[6] 2023. *Memory leak in Binutils.* Retrieved 2023-05-05 from https://sourceware.org/bugzilla/show_bug.cgi?id=29925
[7] 2023. *Peach.* Retrieved 2023-05-05 from https://gitlab.com/peachtech/peach-fuzzer-community
[8] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. 2019. REDQUEEN: Fuzzing with Input-to-State Correspondence.. In *NDSS*, Vol. 19. 1–15.
[9] Fabrice Bellard. 2005. QEMU, a fast and portable dynamic translator.. In *USENIX annual technical conference, FREENIX Track*, Vol. 41. Califor-nia, USA, 46.
[10] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2016. Coverage-based greybox fuzzing as markov chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. 1032–1043.
[11] Juan Caballero, Heng Yin, Zhenkai Liang, and Dawn Song. 2007. Polyglot: Automatic extraction of protocol message format using dynamic binary analysis. In *Proceedings of the 14th ACM conference on Computer and communications security*. 317–329.
[12] Oliver Chang, Jonathan Metzman, Max Moroz, Martin Barbella, and Abhishek Arya. 2016. OSS-Fuzz: Continuous Fuzzing for Open Source Software. *URL: https://github. com/google/ossfuzz* (2016).
[13] Peng Chen and Hao Chen. 2018. Angora: Efficient fuzzing by principled search. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 711–725.
[14] Peng Chen, Jianzhong Liu, and Hao Chen. 2019. Matryoshka: fuzzing deeply nested branches. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 499–513.
[15] Weidong Cui, Jayanthkumar Kannan, and Helen J Wang. 2007. Discoverer: Automatic Protocol Reverse Engineering from Network Traces.. In *USENIX Security Symposium*. 1–14.
[16] Weidong Cui, Marcus Peinado, Karl Chen, Helen J Wang, and Luis Irun-Briz. 2008. Tupni: Automatic reverse engineering of input formats. In *Proceedings of the 15th ACM conference on Computer and communications security*. 391–402.
[17] Andrea Fioraldi, Daniele Cono D'Elia, and Emilio Coppa. 2020. WEIZZ: Automatic grey-box fuzzing for structured binary formats. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 1–13.
[18] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. {AFL++}: Combining incremental steps of fuzzing research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*.
[19] Shuitao Gan, Chao Zhang, Peng Chen, Bodong Zhao, Xiaojun Qin, Dong Wu, and Zuoning Chen. 2020. GREYONE: Data Flow Sensitive Fuzzing.. In *USENIX Security Symposium*. 2577–2594.
[20] Vivek Jain, Sanjay Rawat, Cristiano Giuffrida, and Herbert Bos. 2018. TIFF: using input type inference to improve fuzzing. In *Proceedings of the 34th Annual Computer Security Applications Conference*. 505–517.
[21] Min Gyung Kang, Stephen McCamant, Pongsin Poosankam, and Dawn Song. 2011. Dta++: dynamic taint analysis with targeted control-flow propagation.. In *NDSS*.
[22] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *International symposium on code generation and optimization, 2004. CGO 2004.* IEEE, 75–86.
[23] JongHyup Lee, Thanassis Avgerinos, and David Brumley. 2011. TIE: Principled reverse engineering of types in binary programs. (2011).
[24] Yuwei Li, Shouling Ji, Yuan Chen, Sizhuang Liang, Wei-Han Lee, Yueyao Chen, Chenyang Lyu, Chunming Wu, Raheem Beyah, Peng Cheng, et al. 2021. UNIFUZZ: A Holistic and Pragmatic Metrics-Driven Platform for Evaluating Fuzzers.. In *USENIX Security Symposium*. 2777–2794.
[25] Jie Liang, Mingzhe Wang, Chijin Zhou, Zhiyong Wu, Yu Jiang, Jianzhong Liu, Zhe Liu, and Jiaguang Sun. 2022. PATA: Fuzzing with path aware taint analysis. In *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1–17.
[26] Zhiqiang Lin, Xiangyu Zhang, and Dongyan Xu. 2010. Automatic reverse engineering of data structures from binary execution. In *Proceedings of the 11th Annual Information Security Symposium*. 1–1.
[27] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: building customized program analysis tools with dynamic instrumentation. *Acm sigplan notices* 40, 6 (2005), 190–200.
[28] Chenyang Lyu, Shouling Ji, Chao Zhang, Yuwei Li, Wei-Han Lee, Yu Song, and Raheem Beyah. 2019. MOPT: Optimized Mutation Scheduling for Fuzzers.. In *USENIX Security Symposium*. 1949–1966.
[29] Jonathan Metzman, László Szekeres, Laurent Simon, Read Sprabery, and Abhishek Arya. 2021. Fuzzbench: an open fuzzer benchmarking platform and service. In *Proceedings of the 29th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering*. 1393–1403.
[30] M.Zalewski. [n. d.]. American fuzzy lop. http://lcamtuf.coredump.cx/afl/. 2023.
[31] Van-Thuan Pham, Marcel Böhme, Andrew E Santosa, Alexandru Răzvan Căciulescu, and Abhik Roychoudhury. 2019. Smart greybox fuzzing. *IEEE Transactions on Software Engineering* 47, 9 (2019), 1980–1997.
[32] Sebastian Poeplau and Aurélien Francillon. 2020. Symbolic execution with SymCC: Don't interpret, compile!. In *Proceedings of the 29th USENIX Conference on Security Symposium*. 181–198.
[33] Sebastian Poeplau and Aurélien Francillon. 2021. SymQEMU: Compilation-based symbolic execution for binaries.. In *NDSS*.
[34] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. {AddressSanitizer}: A fast address sanity checker. In *2012 USENIX annual technical conference (USENIX ATC 12)*. 309–318.
[35] Ji Shi, Zhun Wang, Zhiyao Feng, Yang Lan, Shisong Qin, Wei You, Wei Zou, Mathias Payer, and Chao Zhang. [n. d.]. AIFORE: Smart Fuzzing Based on Automatic Input Format Reverse Engineering. ([n. d.]).
[36] Asia Slowinska, Traian Stancescu, and Herbert Bos. 2011. Howard: A Dynamic Excavator for Reverse Engineering Data Structures.. In *NDSS*.
[37] Rui Wang, XiaoFeng Wang, Kehuan Zhang, and Zhuowei Li. 2008. Towards automatic reverse engineering of software security configurations. In *Proceedings of the 15th ACM conference on Computer and communications security*. 245–256.
[38] Tielei Wang, Tao Wei, Guofei Gu, and Wei Zou. 2010. TaintScope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection. In *2010 IEEE Symposium on Security and Privacy*. IEEE, 497–512.
[39] Website. [n. d.]. 010 Editor. https://www.sweetscape.com/010editor/. 2022.
[40] Gilbert Wondracek, Paolo Milani Comparetti, Christopher Kruegel, Engin Kirda, and Scuola Superiore S Anna. 2008. Automatic Network Protocol Analysis.. In *NDSS*, Vol. 8. Citeseer, 1–14.
[41] Meng Xu, Sanidhya Kashyap, Hanqing Zhao, and Taesoo Kim. 2020. Krace: Data race fuzzing for kernel file systems. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1643–1660.
[42] Wei You, Xuwei Liu, Shiqing Ma, David Perry, Xiangyu Zhang, and Bin Liang. 2019. SLF: Fuzzing without valid seed inputs. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 712–723.
[43] Wei You, Xueqiang Wang, Shiqing Ma, Jianjun Huang, Xiangyu Zhang, XiaoFeng Wang, and Bin Liang. 2019. Profuzzer: On-the-fly input type probing for better zero-day vulnerability discovery. In *2019 IEEE symposium on security and privacy (SP)*. IEEE, 769–786.
[44] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. 2018. QSYM: A practical concolic execution engine tailored for hybrid fuzzing. In *27th USENIX Security Symposium (USENIX Security 18)*. 745–761.