FormatFuzzer: Effective Fuzzing of Binary File Formats

RAFAEL DUTRA, CISPA Helmholtz Center for Information Security, Germany
RAHUL GOPINATH, CISPA Helmholtz Center for Information Security, Germany
ANDREAS ZELLER, CISPA Helmholtz Center for Information Security, Germany

Effective fuzzing of programs that process structured binary inputs, such as multimedia files, is a challenging task, since those programs expect a very specific input format. Existing fuzzers, however, are mostly *format-agnostic,* which makes them versatile, but also ineffective when a specific format is required.

We present FormatFuzzer, a generator for *format-specific fuzzers.* FormatFuzzer takes as input a *binary template* (a format specification used by the 010 Editor) and compiles it into C++ code that acts as *parser, mutator,* and highly efficient *generator* of inputs conforming to the rules of the language.

The resulting format-specific fuzzer can be used as a standalone producer or mutator in black-box settings, where no guidance from the program is available. In addition, by providing mutable *decision seeds,* it can be easily *integrated* with arbitrary *format-agnostic fuzzers* such as AFL to make them format-aware. In our evaluation on complex formats such as MP4 or ZIP, FormatFuzzer showed to be a highly effective producer of valid inputs that also detected previously unknown memory errors in `ffmpeg` and `timidity`.

CCS Concepts: • **Software and its engineering → Software testing and debugging**; *Translator writing systems and compiler generators*; *Parsers*; *Syntax*.

Additional Key Words and Phrases: structure-aware fuzzing, file format specifications, binary files, grammars, parser generators, generator-based fuzzing

1 INTRODUCTION

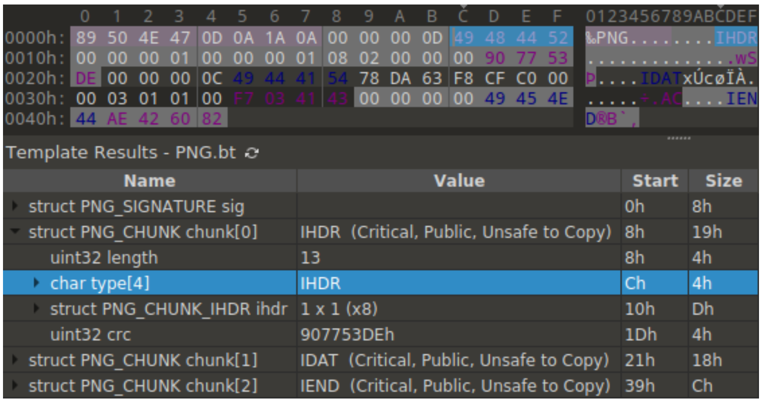
Feedback-directed fuzzing tools such as AFL [58] and libFuzzer [43] have shown great success in finding bugs and vulnerabilities in widely used software. Such tools are generally *format-agnostic*, meaning that they assume no knowledge about the input format when producing or mutating inputs. This makes them easy to set up and deploy. However, it also limits their usage to settings in which (1) Sample input files to be mutated are available; (2) Instrumentation and feedback from the program under test is available; (3) The cost of producing myriads of invalid inputs is bearable.

The alternative to a format-agnostic fuzzer is to create a *format-specific* fuzzer, making use of format knowledge to produce valid inputs. However, creating such a fuzzer comes with considerable effort, in particular when one wants to reuse the guidance and analysis capabilities of feedback-directed fuzzers.

In this paper, we present a novel approach that combines the flexibility of format-agnostic fuzzers with the efficacy of format-specific fuzzers. Our FormatFuzzer framework leverages existing *binary templates*—specifications for input formats, from JPEG to PCAP—to produce or mutate valid file inputs, even in black-box settings where no guidance from the program is available; and integrate with arbitrary *format-agnostic fuzzers* such as AFL to make them format aware. In all settings, using FormatFuzzer *increases coverage in the program under test,* covering code not reached by a format-agnostic fuzzer, and thus increasing the chance of detecting bugs and vulnerabilities.

How does FormatFuzzer work? The inputs for FormatFuzzer are *existing input specifications.* The 010 Editor [47] is a commercial editor which allows users to view and edit binary files with detailed structural information, as shown in Figure 1. To parse its inputs, the editor relies on human written specification files, known as *binary templates* [46] to specify the different sections of a file. Those binary templates are written in a C-like language and have been community-developed for

Authors' addresses: Rafael Dutra, rafael.dutra@cispa.de, CISPA Helmholtz Center for Information Security, Stuhlsatzenhaus 5, Saarbrücken, Germany, 66111; Rahul Gopinath, rahul.gopinath@cispa.de, CISPA Helmholtz Center for Information Security, Stuhlsatzenhaus 5, Saarbrücken, Germany, 66111; Andreas Zeller, zeller@cispa.de, CISPA Helmholtz Center for Information Security, Stuhlsatzenhaus 5, Saarbrücken, Germany, 66111.



```
        0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F   0123456789ABCDEF
0000h: 89 50 4E 47 0D 0A 1A 0A 00 00 00 0D 49 48 44 52  %PNG........IHDR
0010h: 00 00 00 01 00 00 00 01 08 02 00 00 00 90 77 53  .............wS
0020h: DE 00 00 00 0C 49 44 41 54 78 DA 63 F8 CF C0 00  Þ....IDATxÚcøÏÀ.
0030h: 00 03 01 01 00 F7 03 41 43 00 00 00 00 49 45 4E  .....÷.AC....IEN
0040h: 44 AE 42 60 82                                   D®B`‚
```

Template Results - PNG.bt

Name	Value	Start	Size
struct PNG_SIGNATURE sig		0h	8h
struct PNG_CHUNK chunk[0]	IHDR (Critical, Public, Unsafe to Copy)	8h	19h
uint32 length	13	8h	4h
char type[4]	IHDR	Ch	4h
struct PNG_CHUNK_IHDR ihdr	1 x 1 (x8)	10h	Dh
uint32 crc	907753DEh	1Dh	4h
struct PNG_CHUNK chunk[1]	IDAT (Critical, Public, Unsafe to Copy)	21h	18h
struct PNG_CHUNK chunk[2]	IEND (Critical, Public, Unsafe to Copy)	39h	Ch

Fig. 1. 010 Editor displaying a PNG file.

```
typedef struct {
    byte btRed;
    byte btGreen;
    byte btBlue;
} PNG_PALETTE_PIXEL;

struct PNG_CHUNK_PLTE (int32 chunkLen) {
    PNG_PALETTE_PIXEL plteChunkData[chunkLen/3];
};
```

Listing 1. PNG binary template (extract).

over ten years, resulting in a repository of more than 200 different specifications for popular binary file formats [45]. So, for many popular fuzzing targets, a binary template which describes its input format is already available online. For instance, the majority of bugs listed under AFL's bug trophy case [58] come from programs which process popular binary formats for which there is a public specification. Of special interest are multimedia file formats, such as images, audio or video, which often come from untrusted sources, and therefore represent a large attack surface, with potentially serious security consequences.

As an example of a binary template, consider Listing 1, showing an excerpt of the original template for PNG files. We see that (like a C program), it defines *types* such as `PNG_PALETTE_PIXEL` consisting of three byte values. The following `struct` fragment defines a chunk of such pixels named `PNG_CHUNK_PLTE`, whose overall (parameterized) length is `chunkLen`. The full PNG format contains dozens of such chunk definitions, all formalized within the PNG binary template; these also include executable code that computes context-sensitive information such as checksums (whose inference is a major roadblock for format-agnostic fuzzers).

FormatFuzzer takes as input one of these binary templates—say, a PNG template. It then *compiles* it into:

(1) A highly efficient *generator* that produces outputs in the format specified (i.e., a PNG fuzzer); and
(2) A *parser* that reads in existing inputs in the format specified (i.e., a PNG parser).

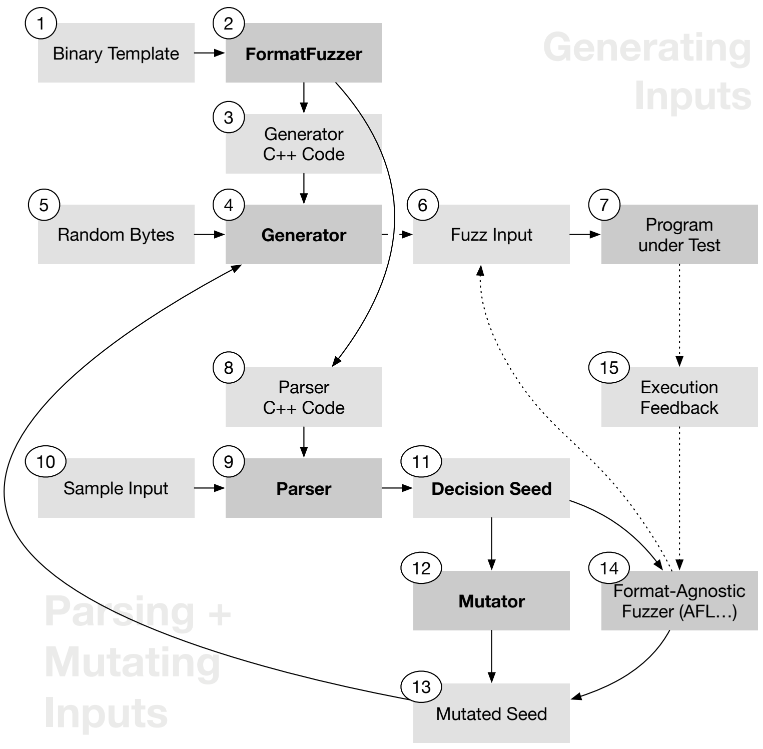


Fig. 2. FormatFuzzer overview. From a *binary template* (1) that specifies an input format F, FormatFuzzer (2) generates C++ code (3) that compiles into a highly efficient *generator* (4) for F. In a *black-box testing* setting, the generator uses a source of random bytes (5) to generate fuzz input (6) in the format F for the program under test (7). From the same binary template, FormatFuzzer also generates C++ code (8) for a *parser* (9) for F. In a *mutation* setting, this parser transforms a sample input (10) into a *decision seed* (11), which encodes the decisions made while parsing the sample into a string of bytes. FormatFuzzer can *mutate* this seed (12) (for example, replacing one chunk); feeding the mutated seed (13) into the generator yields a mutated fuzz input still conforming to F. Finally, for an *evolutionary* setting, the mutator can also be a format-agnostic fuzzer (14) such as AFL, which then mutates and evolves seed strings based on execution feedback (15). Such a fuzzer can also apply out-of-format mutations to the fuzz input directly.

The *parser* can immediately be used in existing *parser-enhanced* fuzzers such as AFLSmart [39], effectively giving us a format-specific fuzzer that evolves, say, given PNG seed inputs; this can be repeated with hundreds of different templates, producing hundreds of format-specific, coverage-guided fuzzers. The *generator,* on the other hand, can be used as a *standalone* generator of inputs that all conform to the format specifications. This is useful in black-box settings, where feedback-driven fuzzers cannot be applied. Furthermore, the combination of *parser* and *generator* allows FormatFuzzer to *mutate* existing inputs in a way that ensures their validity. Finally, FormatFuzzer allows to *integrate format-agnostic fuzzers and make them format-aware,* either by having them apply smart (format-aware) mutations on the inputs, or by having them mutate *decision seeds*—strings of bytes that represent decisions taken during parsing and generating. Both integrations bring together the best of format-agnostic guidance and format-aware mutations, and increase the reach of format-agnostic fuzzers.

Nevertheless, some manual effort is still required. The hundreds of existing binary templates work well for *parsing;* but to make effective *generators,* one has to further refine them. These refinements, however, are easy to write and the feature-full language of binary templates is rich enough to express virtually all needed features of binary file formats, including length fields, checksums, complex structures and constraints between variables. Extending a binary template and compiling it is a one-time effort for each format; and once one has, say, a full-fledged PNG specification, one gets a PNG fuzzer, a PNG parser, a PNG mutator, and a variant of your favorite format-agnostic, feedback-directed fuzzer that produces thousands of valid PNG files per second—with only a fraction of the effort it would take to build such a fuzzer from scratch.

In the remainder of the paper, we detail the techniques behind FormatFuzzer and how they improve the state of the art in fuzzing. Specifically, we make the following contributions:

(1) **A novel description format for generating binary inputs.** There are hundreds of well-curated *binary templates* available that describe just as many binary file formats. In Section 2, we describe their structure; and in Section 3 we extend them with new features for *generating* valid inputs.
(2) **Decision seeds to synchronize generators and parsers.** FormatFuzzer takes binary templates to produce C++ code (Figure 2) that compiles (Section 4) into specialized and highly efficient generators and parsers, including several context-sensitive features (Section 3). Generator and parser synchronize over *decision seeds*—a sequence of bytes reflecting decisions taken during generating or parsing (Section 5). After parsing an input, the resulting decision seed allows the generator to re-take the same decisions (but now for generating), which reproduces the input exactly. As far as we know, FormatFuzzer is the first framework to enable this bidirectional transformation between decision seeds and binary files. This allows us to apply novel smart mutations (Section 6.2) over decision seeds which are obtained from a high-quality corpus of binary files.
(3) **Making format-agnostic fuzzers input-format aware.** Mutating bytes in the decision seed allows to explore the entire domain of inputs. If a format-agnostic fuzzer is thus set to mutate and evolve decision seeds, FormatFuzzer can then translate any such mutation into a valid binary input. This enables a generator-based approach for integrating FormatFuzzer with any format-agnostic fuzzer, leveraging its mutation and evolution strategies (Section 6). In addition, FormatFuzzer provides novel smart mutations, which operate over decision seeds and can therefore respect contextual information in mutated files. This enables another highly effective approach to improve any format-agnostic fuzzer, by leveraging the format-specific *smart mutations* provided by FormatFuzzer (Section 6).
(4) **A detailed evaluation of all FormatFuzzer elements.** We evaluated FormatFuzzer (Section 7) on 10 different programs and file formats (PNG, JPEG, GIF, MIDI, MP4, ZIP, PCAP, AVI, WAV, BMP) and found that:
    (a) FormatFuzzer very quickly produces valid inputs, as standalone generator as well as mutator.
    (b) FormatFuzzer integrates well with format-agnostic fuzzers such as AFL, reaching lines otherwise not covered. In our initial fuzzing experiments, FormatFuzzer detected a number of previously unknown (and potentially exploitable) segmentation faults and aborts in `ffmpeg` and `timidity`.

FormatFuzzer is available as open source and enjoys a quickly growing user community. For details, see https://github.com/uds-se/FormatFuzzer.

2 THE BINARY TEMPLATE LANGUAGE

We start our exploration into FormatFuzzer with a description of its central input—*binary templates.* The *binary template language* [48] was designed to fully specify any binary format. Its syntax and semantics are close to the C programming language, the main difference being that it describes not programs, but data formats. Its main elements are:

Variable (input data) declarations. Every variable declared in the binary template is *directly mapped to a sequence of bytes in the input.* If a template starts with `uint32 len; char s[len]; int64 n;`, this means that the input consists of
(1) a 32-bit unsigned integer (`len`) in the first four bytes;
(2) an array of characters of length `len` in the next `len` bytes; and
(3) a 64-bit integer (`n`) in the next four bytes.
Note that array sizes can be any expression including variables and functions; this allows to specify variable lengths.

Local variables. When a variable is defined with the keyword `local`, it is *not* mapped to input contents, but acts like a traditional variable stored in memory. It can be freely used in computations and control-flow decisions by the binary template.

Type declarations. As in C, one can define *types* using the `typedef` and `struct` constructs. Listing 1 defines a type `PNG_PALETTE_PIXEL`, composed of three individual bytes.
Besides native types (such as `uint32` and `string`) and `struct` types, variables can also have `enum` types, where the list of possible values is explicitly provided.

Parameters. Unlike C, type and variable declarations can take *parameters.* The definition of `PNG_CHUNK_PLTE` in Listing 1, for instance, is parameterized with a `chunkLen` parameter, which sets the size of the contained array.

Conditionals. Binary templates incorporate all the usual control-flow constructs (`if`, `else`, `while`, `switch`, etc.); their bodies would again contain variable declarations. These are used to express *alternatives* and *loops* in the input. For instance, to express a sequence of chunks, whose actual type depends on a header, we could use a loop and conditions as in Listing 2.

```
while (!FEof()) {
    uint32 length;
    char   type[4];
    if (type == "IHDR")
        PNG_CHUNK_IHDR  ihdr;
    else if (type == "PLTE")
        PNG_CHUNK_PLTE  plte(length);
    /* ... */
    if (type == "IEND")
        break; /* break out of while loop */
}
```

Listing 2. Loops and alternatives in a binary template.

Built-in functions. The `while` condition in Listing 2 invokes a *function* named `FEof()` to check for an end-of-file condition. The language provides several such functions to allow for greater control during parsing, making the language expressive enough to support complex constructs from hundreds of binary formats. Examples include `FTell()`, which returns the current file position, `FSeek()`, which allows jumping to a different file position for non-sequential parsing, and `FileSize()`, which returns the file size. Advanced lookahead functions such as `FindFirst()` and `FindAll()` allow searching the file for specific tokens.

Lookahead functions allow the parser to *peek* at a few later bytes in the file without advancing the file position, and can be used for control-flow decisions. For instance, peeking at the first bytes of a `struct` ahead of time can help determine which `struct` type should be parsed. Examples of lookahead functions are `ReadByte()` and `ReadBytes()`.

Besides I/O functions, a multitude of string and math functions is available, including conversion and checksum algorithms.

Custom functions. The language also allows defining own functions, which follow C function syntax and semantics, again allowing variable declarations for describing input data. As all the operators from C are also supported, this makes binary templates a *Turing-complete description of input formats.*

Error handling. In the case of irrecoverable parsing errors, the language also allows early termination, using a top-level `return` statement.

A full definition of syntax and semantics of the binary template language is available online [48].

We would like to note that our choice for binary templates as the base for FormatFuzzer is purely pragmatic. Binary templates have been shown to be able to precisely document hundreds of formats; they are reasonably well documented; and as we will show in this paper, they are not too hard to adapt for effective fuzzing. Whether all the subtleties of input formats can be specified in a more elegant or useful way is yet to be proven.

3 BINARY TEMPLATES FOR GENERATION

In order to use binary templates to *generate* new files from scratch, some decisions need to be made during the generation process. Most notably, a value will have to be chosen for every variable that is declared, and also for every lookahead function that is called. If such values were to be chosen uniformly at random from the entire range of possible values (for example, 2^{32} possible values for a `uint32`), most variables would end up having nonsensical values, rendering the file invalid. Therefore, we extend the language of binary templates in a number of aspects:

Choices of valid values. Our first extension allows the specification of an *array of valid values to choose from.* When a variable is declared, this array of choices can be assigned as an *initialization list.* Line 11 of Listing 3 tells the generator to pick a value for `bits` uniformly from five possible choices.

Enumerations. To handle enumerations, the generator chooses a value uniformly from the set of the enumerated values (e.g., `PNG_COLOR_SPACE_TYPE` in Listing 3), which is usually the appropriate behavior. But it is also possible to specify a different set of choices through an initialization list.

Lookaheads. For lookahead functions, we extend the language to allow a new argument specifying the possible values to choose from. One example is the array `colors` passed as an argument to the `ReadByte()` call in Line 9 of Listing 3.

Allowing invalid choices. Finally, we also allow the generator to deviate from the specified behavior and pick any value from the entire range of 2^8 possible values for `ubyte` with a small probability of about 1%, by performing an action that we call an "evil" decision. Such evil decisions are enabled by default, but can be turned on and off during generation by calling a function `SetEvilBit()`, which takes as input the new value for the flag and returns the old value. When enabled, they can give the fuzzer greater diversity of generated files by allowing the fuzzer to explore a small number of invalid choices during generation.

Let us now put these extensions into action and show what changes had to be made to the binary template in order to support the generation of valid inputs. Our running example, PNG, is representative of the changes that are required for other file formats as well. This section

```
typedef enum <byte> {
    GrayScale=0, TrueColor=2, Indexed=3, AlphaGrayScale=4, AlphaTrueColor=6
} PNG_COLOR_SPACE_TYPE;

typedef struct {
|   uint32  width <min=1, max=24>;
|   uint32  height <min=1, max=24>;
|   local byte colors[] = { GrayScale, TrueColor, Indexed, /* ... */ };
|   switch (ReadByte(FTell() + 1, colors)) { /* color_type */
|   case GrayScale:
|       ubyte   bits = { 1, 2, 4, 8, 16 };
|       break;
|   case TrueColor:
|       ubyte   bits = { 8, 16 };
|       break;
|   case Indexed:
|       ubyte   bits = { 1, 2, 4, 8 };
|       break;
|   /* ... */
|   }
    PNG_COLOR_SPACE_TYPE color_type;
    PNG_COMPR_METHOD   compr_method;
    PNG_FILTER_METHOD  filter_method;
    PNG_INTERLACE_METHOD interlace_method;
} PNG_CHUNK_IHDR;
```

Listing 3. Contents of the IHDR chunk, where bits depends on color_type. Changes made to enable generation are highlighted in blue.

exhaustively describes all the changes which were needed to generate valid PNGs. For any format we have seen so far, all its modifications also fall into the classes described here. All our changes are highlighted in blue in the code listings.

3.1 Magic Values

The simplest common feature of binary formats is the use of *magic values*, where certain bytes in the file need to have a specific value. For example, a PNG file needs to start with a signature containing the bytes `"\x89504E470D0A1A0A"`. In the original PNG template, there was already a check which compares the signature to the expected value and terminates the template if the signature is invalid, as shown in Listing 4. In this case, FormatFuzzer is able to automatically mine such comparisons (with operators != or ==) when analyzing the source code of the binary template and remember the magic values that should be used. For example, the value `0x8950` will be remembered as a good value to use for index `0` in the array `btPngSignature`. Such comparisons can also be mined inside struct definitions and functions.

The only change made to the binary template for the signature generation was adding two calls to `SetEvilBit()`, which disable evil decisions temporarily for the signature generation, re-enabling them afterwards. This essentially marks the generation of the `sig` variable as *strict*, which is useful because any file with an incorrect PNG signature is not a valid PNG file and would be immediately rejected by the target program.

```
typedef struct {
   uint16 btPngSignature[4];
} PNG_SIGNATURE;

local int evil = SetEvilBit(false);
PNG_SIGNATURE sig;
SetEvilBit(evil);
if (sig.btPngSignature[0] != 0x8950 ||
       sig.btPngSignature[1] != 0x4E47 ||
       sig.btPngSignature[2] != 0x0D0A ||
       sig.btPngSignature[3] != 0x1A0A) {
   Warning("File is not a PNG image.");
   return -1;
}
```

Listing 4. PNG signature: the magic bytes are mined automatically.

3.2 Size Fields

Another common feature in binary formats is the presence of size fields, whose value correspond to the size of a certain structure in the file, or some particular position or index inside the file. Such size fields are context-sensitive features, and binary formats using such features are not expressible using context-free grammars. It is particularly crucial that size fields are generated correctly, because a wrong value for a size field would completely change the interpretation of the remaining bytes of the file.

In the PNG format, the chunk `length` is the first field in every chunk, as seen in Line 2 of Listing 5. There, we have extended the definition of the `length` variable with additional metadata `min=1` and `max=16` to restrict the choices of values for this variable to a small range. This is important since this variable will be used as the length of an array, as seen in Line 11 of Listing 5. If no such bounds are specified, FormatFuzzer still samples values for integer variables from a skewed distribution that prioritizes small positive integers, but still allows arbitrary large or negative integers with a smaller probability.

One challenge in generating PNG chunks which is also common in other file formats is that the `length` field appears before the data it refers to. And it is not always possible to predict beforehand what the length of certain data should be before generating that data. In addition, a bad choice for `length` could lead to the length of some array becoming too large or negative, such as when an integer underflow happens in Line 4 of Listing 6. The strategy we use to tackle those issues is to use the initial value sampled for `length` only as a hint. So if, for example, this integer underflow happens when defining the array `frame_data`, we ignore the value of `length` and choose some small positive integer to use as the length of the array. This behavior can be enabled in the binary template with a call to the function `ChangeArrayLength()`.

Now, since the chosen value of `length` is used only as a hint, the generated data may end up being larger or smaller than expected. This needs to be corrected. Otherwise the chunk will have a different length than what is specified in the `length` variable, leading to all following chunks being interpreted incorrectly. Therefore, after generating the data inside a chunk, we check if the length is correct. In case of a discrepancy, we go back and overwrite the value of `length` with the correct value, as shown in Lines 13 to 20 of Listing 5. There, the `FSeek()` function is used to jump to a different position in the file. Note that we temporarily disable evil decisions when fixing the value of `length`, because any wrong value would completely change the file parse tree. Such common

```
 1   typedef struct {
 2 |     uint32 length <min=1, max=16>;
 3     local int64 start = FTell();
 4     char   type[4];
 5     if (type == "IHDR")
 6         PNG_CHUNK_IHDR  ihdr;
 7     else if (type == "PLTE")
 8         PNG_CHUNK_PLTE  plte(length);
 9     /* ... */
10 |     else if (length > 0 && type != "IEND")
11         ubyte  data[length];
12     local int64 end = FTell();
13 |     local uint32 correct = end - start - 4;
14 |     if (length != correct) { /* Fix it */
15 |         FSeek(start - 4);
16 |         local int evil = SetEvilBit(false);
17 |         uint32 length = { correct };
18 |         SetEvilBit(evil);
19 |         FSeek(end);
20 |     }
21     local uint32 crc_calc = Checksum(CHECKSUM_CRC32, start, end-start);
22 |     uint32 crc = { crc_calc };
23     if (crc != crc_calc)
24         Warning("Bad CRC %
25 } PNG_CHUNK;
```

Listing 5. Definition of a PNG chunk, where we can see the changes to correct the chunk `length` and `crc`.

```
| local uint32 seq_num = 0;
struct PNG_CHUNK_FDAT {
|     uint32 sequence_number = { seq_num++ };
    ubyte  frame_data[length-4];
};
```

Listing 6. Sequence numbers require a global counter.

size fixes can also be simplified by the use of a macro `FIX_VARIABLE()`, so that this whole block of code can be replaced by a call to the macro.

```
// arguments: type ,  name ,  correct value , position
FIX_VARIABLE(uint32, length, end - start - 4, start - 4);
```

3.3 Complex Constraints

Next we discuss how to satisfy some more complex constraints between variables in the template.

Relationship between variables. One common pattern happens when the value of one variable `x` restricts the possible values for another variable `y`. If `x` is generated first, then we can simply use the value of `x` in control-flow decisions or initialization lists that will define how `y` is generated. For example, a PNG chunk of type `"IEND"` must have length 0. This can be ensured by using the value of `type` in a control-flow decision in Line 10 of Listing 5 to ensure no `data` will be generated if `type` is `"IEND"`.

In case x will only be generated after y, we can still look first at the value of x using a lookahead function. Listing 3 shows the implementation of the IHDR chunk, where the possible values for the bits variable are restricted by the value of the next variable color_type. By using the lookahead function ReadByte() in Line 9, we can first look at the value at position FTell()+1, one byte after the current position. This will be the value of color_type, and can be used in a switch to determine the possible values for bits. In generation mode, ReadByte() will use the array colors provided as an argument to choose which value should be generated at position FTell()+1 in the file. Once a value is chosen by the lookahead function, it is fixed and cannot be overwritten. Therefore, when color_type is declared later in Line 21, it is guaranteed to have the same value that was previously chosen by ReadByte().

Checksums. Checksums are a common context-sensitive feature of binary formats. Luckily, the binary template language already provides a helper function Checksum() which implements common checksum algorithms. Therefore, checksums can be handled simply by computing the correct checksum value and specifying it as the desired value for the checksum variable, as shown in Line 22 of Listing 5.

Global state. Some features, such as sequence numbers which are incremented every time, require some global state to be generated correctly. This can be handled by defining a local variable of global scope, as in Listing 6.

3.4 Chunk Ordering

A big challenge in generating valid files is choosing a valid sequence of chunk types. The set of possible types for the next chunk is usually context-sensitive, depending on which chunks have already been generated. In PNG, for instance, the chunks IHDR, IDAT and IEND are mandatory. The first chunk must be IHDR and the last chunk must be IEND. Some optional chunks, such as tIME, can appear before or after IDAT. Other optional chunks, such as bKGD, can only appear before IDAT.

If we are only interested in parsing (not generating) PNG files, the following implementation suffices to parse a sequence of chunks.

```
while (!FEof())
    PNG_CHUNK chunk;
```

However, to generate a *valid* sequence of PNG chunks, we need a way to specify the set of possible choices and allow those choices to change for each new generated chunk.

To solve this problem, we have extended the lookahead function ReadBytes() with 3 new arguments: an array of preferred values, an array of possible values and a probability to pick from the preferred values. We have also added new operators -= and += which can add or remove values from such arrays. Listing 7 shows a binary template that can generate a sequence of PNG chunks satisfying the ordering constraints. Here, function ReadBytes() in Line 5 needs to choose the 4-byte value which will be written to position FTell()+4 in the file, which is the location of the type array on the next chunk. The chosen value is also returned in the chunk_type variable, which is passed by reference to the function.

The preferred array is used to specify what is the next mandatory chunk, while the possible array includes all chunks which could be inserted in the next position. With a probability of 0.25, ReadBytes() will try to pick a value from preferred, to ensure that it makes progress towards finishing the generation and does not spend too much time creating lots of optional chunks. With the complementary probability of $1 - 0.25$, ReadBytes() will pick a value from the possible array (or even a completely random value if an evil decision is made). If the preferred array is empty,

```
 1 | ChangeArrayLength();
 2 | local char chunk_type[4];
 3 | local string preferred[] = { "IHDR" };
 4 | local string possible[] = { "IHDR" };
 5 | while (ReadBytes(chunk_type, FTell() + 4, 4, preferred, possible, 0.25)) {
 6       PNG_CHUNK chunk;
 7 |     switch (chunk_type) {
 8 |     case "IHDR":
 9 |         switch (chunk.ihdr.color_type) {
10 |         case GrayScale:
11 |             local string preferred[] = { "IDAT" };
12 |             local string possible[] = { "tIME", "tEXt", "pHYs", /*...*/ "bKGD", "IDAT" };
13 |             break;
14 |         /* ... */
15 |         }
16 |         break;
17 |     case "IDAT":
18 |         local string preferred[] = { "IEND" };
19 |         possible -= ("IDAT", "pHYs", /*...*/ "bKGD");
20 |         possible += "IEND";
21 |         break;
22 |     /* ... */
23 |     case "IEND":
24 |         local string preferred[0];
25 |         local string possible[0];
26 |         break;
27 |     }
28 | }
```

Listing 7. Use of ReadBytes() to define chunk ordering.

ReadBytes() can choose a value from possible, but it is also allowed to not choose any value, not write any value to the file and return false, leaving the while loop. This signals that the file is now complete and no more chunks will be added.

With no evil decisions, a valid chunk order is guaranteed. The first chunk must be IHDR as the only option specified in Lines 3 and 4. If an IHDR is generated with color_type equal to GrayScale, the next preferred chunk will be IDAT (Line 11), with several possible chunks available (Line 12), including tIME and bKGD. After IDAT is generated, IEND will be the next preferred chunk (Line 18) and the possible array is updated with operators -= and += in Lines 19 and 20 to remove and add new values. For instance, bKGD is removed because a bKGD chunk is not allowed after IDAT. Finally, when an IEND is generated, both arrays become empty (Lines 24 and 25) to signal that ReadBytes() must return false.

Our implementation of ReadBytes() allows the same while loop from Listing 7 to work for both parsing and generation. In parsing mode, we can easily check whether the bytes read from the file belong to the preferred array, the possible array or to neither (for example, if the position of those bytes would be beyond the end of the file). This allows us to successfully reconstruct the random decisions that would need to be made to generate such a file. Our approach of using ReadBytes() to define chunk ordering turned out to be extremely general, being used in five of the developed formats (PNG, JPEG, MP4, ZIP, AVI).

3.5 Compressed and Encoded Data

With all the changes shown so far, our modified binary template is able to generate almost all chunks of a PNG file correctly. Only the IDAT chunk is still a challenge, because it contains a zlib-compressed datastream of the image pixel data. In order to handle compressed (or otherwise encoded) data, the compression and decompression functions should be specified such that the generator can generate the uncompressed data first and then apply the compression function to it to obtain the datastream that will be written to the file. We have manually implemented this strategy for the PNG IDAT chunk by calling the appropriate functions from the zlib library, allowing FormatFuzzer to generate fully valid PNGs. PNG was the only format so far to require this special handling for compression, because interestingly we found that in other formats simply generating random bytes for the compressed datastreams was sufficient to obtain streams that could be successfully decompressed with high probability.

4 IMPLEMENTATION

In the domain of language-specific fuzzers, FormatFuzzer is a *generator compiler,* bringing together the versatility of language specifications and the efficiency of compiled generator code. FormatFuzzer *compiles* a binary template to C++ code for generating and parsing inputs in the format specified by the template, leveraging existing libraries for parsing [13] and processing [12] binary templates. The generated C++ code for a given format can be compiled into a standalone executable or a shared library that can be loaded by other fuzzers, such as AFL++ [19]. Each `struct` definition in the binary template is implemented as a C++ class. We also create C++ classes to handle native types, such as `uint32` and arrays. Every time a variable is declared in the template, this triggers a call to the `generate()` method of the corresponding class. FormatFuzzer supports all important features of binary templates, including indexing into previous instances of a given variable, and recursive structs, where one `struct` can contain another `struct` of the same type. We support integer variables in big endian and little endian modes, as well as bitfield variables, which can occupy only fractions of a byte.

Our implementation allows setting a maximum size in bytes for the generated file. This is important for fuzzing because smaller files are both faster to produce and to process with the target program. In our experiments, we have set the maximum file size to 64 kB, since we found that even with only a few kilobytes the files can already exhibit the whole variety of chunk types and structures inside them. During generation, any attempt to generate too much data, such as defining an array with a huge length, will throw an exception, aborting the generation. Generation is also aborted if the generator attempts to consume more bytes than available in the decision seed or performs some invalid operation, such as accessing a non-existing field from a `struct` (this can happen for some formats, since control-flow decisions can determine which fields are generated inside a `struct`).

We have also integrated into FormatFuzzer quality assurance tests to verify the correct behavior of our generators and parsers. We use round-trip tests to make sure that whenever a file is successfully generated, then it can also be correctly parsed, and vice versa. We also ensure that our fuzzers can correctly parse a corpus of real files from each format, which were manually collected from GitHub.

5 DECISION SEEDS

The methodology followed by FormatFuzzer is the following: To generate a file, we first provide an array of random bytes, called the *decision seed.* This array is the source of randomness for generator decisions. Such decision seeds can also be reconstructed by FormatFuzzer while *parsing.*

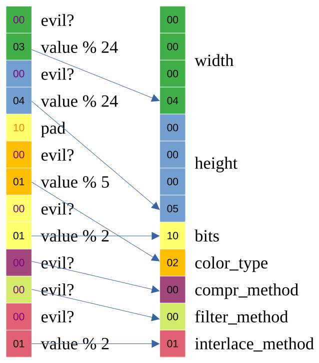


Fig. 3. Decision seed and the corresponding generated content for PNG_CHUNK_IHDR (defined in Listing 3).

5.1 Using Decision Seeds for Generation

Every time the generator needs to make a random decision, such as which value should be used for a variable or lookahead function, it will read the required number of bytes from the decision seed. For example, Figure 3 shows one possible decision seed and the resulting contents of the generated file for a PNG_CHUNK_IHDR (defined in Listing 3). For each of the seven generated variables in this chunk, one decision byte is used to choose whether to make an evil decision. If byte b_0 is read, an evil decision will be taken if $b_0 \bmod 128 = 127$, which happens with probability $1/128$. In the most common case, no evil decision is made. For variables width and height, which were declared with <min=1, max=24>, this means that the next byte b_1 will be used to compute $b_1 \bmod 24$ in order to choose uniformly from one of the 24 possible values.

For the bits and color_type variables, the ReadByte() call in Line 9 of Listing 3 tells us to first skip the bits variable and pick a suitable value for color_type from the array colors, which are the valid values for a PNG_COLOR_SPACE_TYPE. In order to do that, we first consume one decision byte that will be used as a temporary padding for the byte we skipped in the file when jumping to FTell()+1. Then, the next two decision bytes are used to pick the value at position FTell()+1 (which will be color_type): one to decide whether to make an evil decision and one to pick an appropriate value from the five possible choices in the array colors. If we pick the second value in the array, which is TrueColor with value 2, then Line 14 of Listing 3 tells us that there are two possible values to pick for the bits variable: 8 or 16. Finally, when we reach Line 21, which defines the color_type variable, we do not need to consume any more decision bytes, because its value had already been chosen at the previous call to ReadByte().

Given a decision seed, the generation process is completely deterministic. The decision seed encodes the essential features of a given file in a representation which is tailored for fuzzing by exhaustively exploring the file structure. The correspondence between the decision seeds and the generated files can be thought of as a partial function $f : \mathit{Seeds} \nrightarrow \mathit{Files}$. For some values of the seed, the generation can fail. For example, say we have taken an evil decision such as an invalid color_type in the PNG IHDR chunk. This makes it impossible to later generate the data for another chunk. But if the generation succeeds, it associates the seed s to a unique well-defined file $f(s)$. The resulting file $f(s)$ is not guaranteed to conform to the format specification (for instance, due to evil decisions), but it should be valid with high probability. This mapping f is not injective, but ideally every valid file should be obtainable by the generator, i.e. $\mathit{Valid} \subseteq f(\mathit{Seeds})$. This completeness property can be experimentally checked by attempting to parse valid files – say, downloaded from the Internet – with FormatFuzzer. As discussed in the next section, whenever parsing succeeds, we also obtain a decision seed for the parsed file, which in turn means that this file can be the output of the FormatFuzzer generator by taking the appropriate decisions.

5.2 Creating Decision Seeds During Parsing

When FormatFuzzer runs in parsing mode, we obtain not only the parse tree for the input file, but also a *decision seed* that can be used to generate the same input file, as presented in the diagram of Figure 2. This is done by reconstructing at each step the decision bytes that would be required to produce the target file contents. For example, in Figure 3, the decision bytes on the left are exactly the ones that would be obtained by parsing the file contents on the right. For each variable, we find out if an evil decision is required by checking whether its value is one of the possible values specified. In the most common case, no evil decision is required and we can reconstruct the next decision byte as the index of the value that should be picked from the array of possible values. This reconstructed decision seed is not unique. For example, there are 24 valid values for the variables `width` and `height`, so any decision byte which is in the same congruence class modulo 24 would result in the same value.

One important axiom of our framework is that a file can be successfully parsed by FormatFuzzer if and only if the file can be generated from some appropriate decision seed. And the generator and parser must always agree on the same parse tree for the file. That is why we cannot allow evil decisions when fixing the `length` field in Line 17 of Listing 5. If the generator were to overwrite `length` with an evil value, the generator and the parser would disagree about the starting position of the next chunk. Generally, the variables which define sizes or positions in the file, as discussed in Section 3.2, are the only ones which must be generated strictly without evil decisions. Our round-trip tests have shown that generated files can be correctly parsed (and vice versa) even when all other variables are allowed to have evil values.

6 FUZZING STRATEGIES

The FormatFuzzer generators and parsers can be used for numerous fuzzing strategies. We discuss a few in this section.

6.1 Generating Random Files

The simplest approach to fuzzing with FormatFuzzer is to use completely random seeds (say, from `/dev/urandom`) to generate files in a black-box manner. A good fraction of the generated files will be valid according to the format specification and the files will come from a *semantically diverse distribution* in terms of covered features. For example, as discussed in Listing 3, all possible values for `color_type` will be chosen with equal probability, as will the possible values of `bits` for a given `color_type`.

6.2 Mutating Inputs

Another fuzzing strategy enabled by FormatFuzzer is the use of *smart mutations,* where chunks can be abstracted, replaced, deleted or inserted into a file, as seen in Figure 4. Here, we use ‘chunk’ generically to refer to any `struct` or variable defined in the binary template. We define novel smart mutation operations which work over decision seeds, thus allowing contextual information to be taken into account when producing the mutated file.

6.2.1 Smart Abstraction. The first operator we discuss is *smart abstraction*, which allows abstracting one specific chunk c_1 into a new random version c_2. This operation works by first parsing the original

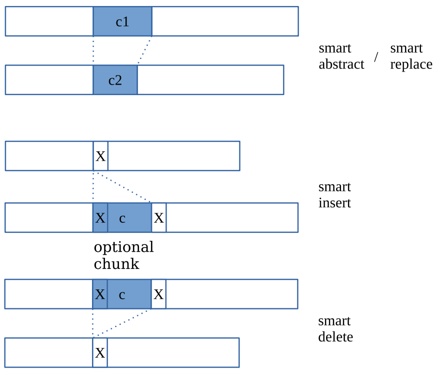


Fig. 4. Smart mutations in FormatFuzzer. Here, X indicates the use of lookahead functions.

file and obtaining its decision seed. Then, we generate a new file by repeating all the decisions that were made before the target chunk c_1 and then using random bytes from `/dev/urandom` to produce the target chunk. Once we detect that the target chunk has been generated, we go back to using the same decision bytes that were used after chunk c_1 in the original file. We have found that smart abstractions were the most useful in reaching new coverage, and they are only possible because of our FormatFuzzer framework, which enables synchronized generation and parsing.

6.2.2 Smart Replacement. The next operator *smart replacement*, which replaces a chunk c_1 from $file_1$ with a different chunk c_2 of the same type from $file_2$. Again, we first parse both files to obtain the decision seeds $seed_1$ and $seed_2$. Then, a new mutated seed is created by replacing the decision bytes which created chunk c_1 with the decision bytes that create chunk c_2, leaving the remaining parts of the seed unchanged, as depicted in the *smart replace* operation of Figure 4. This mutated seed is then used by the generator to produce the mutated file.

To see why performing these operations on the decision seeds can help to generate valid files, consider the case where c_1 is some internal field of a PNG chunk. If c_1 is replaced by a new instance c_2 of the same type, but a different value and different size, then the generator will still compute the correct checksum and size for the modified chunk, while a simple copying of the contents of c_2 into c_1 would leave the checksum and size invalid. This is especially important for successful smart mutations of formats such as MP4, which consist of structs called *boxes* which recursively contain other *boxes*.

6.2.3 Smart Deletions. Our *smart delete* operation consists of deleting a chunk from a file, by removing the decision bytes which were responsible for the generation of this chunk. Such a deletion operation only makes sense if we identify that the target chunk c is optional. We define a chunk c to be *optional* when, right before this chunk is generated, the template calls a lookahead function. This signals that, depending on the result of this lookahead call, we might decide not to generate c at all. For example, the variable `chunk` declared in Line 6 of Listing 7 is considered optional, because its declaration comes after a call to `ReadBytes()`. We allow smart deletion to be performed over a chunk c only when a lookahead function has been called right before and right after the generation of c, as shown in Figure 4. This way, with the new mutated seed, the call to the lookahead function will consume the bytes that were originally consumed at the lookahead call that came after chunk c.

6.2.4 Smart Insertions. Conversely, we define a *smart insert* as the inverse of the *smart delete* operation, trying to insert a new chunk *c* into a file. Here, the original file must have a call to a lookahead function at the insertion position, so that the result of this call could be used to decide whether chunk *c* should be created. The inserted chunk *c* must also be optional. When performing smart insertions and replacements, FormatFuzzer also checks whether the correct number of decision bytes has been consumed in the generation of the new chunk. In case the mutations do not work well because the new chunk does not fit in the desired position, this can be identified and reported by FormatFuzzer. For example, attempting to copy a bKGD chunk (which specifies a background color) will not work if one file uses color type `TrueColor`, which requires three values, while the other uses `GrayScale`, which requires only one.

6.2.5 Cross-File Operations. FormatFuzzer implements procedures to parse a list of files and remember all the information about the chunks so they can be used later for smart mutations. There is also a procedure to apply one smart mutation to a chosen file. Here, we randomly choose which kind of mutation will be applied and which chunks will be involved.

6.3 Integrating with Format-Agnostic Fuzzers

Besides generation and mutation, FormatFuzzer can also integrate with existing *format-agnostic* fuzzers such as AFL [58] in different ways. For our experiments, we have integrated FormatFuzzer with AFL++ [19] version 2.60c as follows.

AFL+FFGen uses AFL to mutate and evolve the *decision seeds*, which will be later fed into the FormatFuzzer generator to produce inputs for the target program. AFL can use the coverage feedback from the program to learn how to effectively mutate the decision seeds. The advantage of working on such seeds is that they are simpler than binary files, since each byte corresponds to a unique decision, and those decisions are made sequentially in the order in which they appear in the seed. Since the FormatFuzzer generator already takes care of the correct input structure, such as computing the correct checksums, inserting the appropriate magic values and setting the correct size fields, AFL can focus exclusively on the high-level decisions that represent the file.

AFL+FFMut lets AFL mutate the files that will be given to the target program as usual, but also adds new mutation operations which are the smart mutations provided by FormatFuzzer. Every interesting input, which achieves new coverage, is saved in the AFL queue. When such an input is about to be mutated for fuzzing, we also parse this input with FormatFuzzer, allowing it be used for smart mutations.

7 EVALUATION

Our evaluation focuses on the following research questions:

RQ1 How much effort is it to set up a binary template file for input generation?
RQ2 How *efficient* is FormatFuzzer in producing inputs?
RQ3 How *accurate* is FormatFuzzer in producing inputs?
RQ4 How accurate are the smart mutations applied by FormatFuzzer?
RQ5 How efficient is FormatFuzzer as a standalone black-box fuzzer?
RQ6 How efficient is the integration of FormatFuzzer with a format-agnostic fuzzer?
RQ7 How does FormatFuzzer compare against other format-aware fuzzers?
RQ8 Does FormatFuzzer find real bugs?

To answer these questions, we ran a series of experiments. All our experiments were conducted on an Intel Xeon CPU E5-4650L machine with 64 cores and 756 GB RAM, running Debian 10. The

Table 1. Number of lines of code required for each format.

Format	Original	Modified Template	Changes		Generated C++
GIF	204	234	+38	−8	2,144
BMP	138	249	+130	−19	1,267
PCAP	220	268	+72	−24	1,388
MIDI	261	282	+39	−18	1,771
AVI	304	471	+248	−81	2,645
PNG	388	492	+114	−10	2,664 (+170)
WAV	574	649	+108	−33	2,457
ZIP	669	774	+191	−86	2,608
MP4	807	1,420	+771	−158	4,556
JPG	1,631	1,662	+282	−251	5,683

fuzzers were run on a single processor for 24 hours. We have repeated each experiment for a total of 10 runs.

In our evaluation, we compare FormatFuzzer against the format-agnostic fuzzer AFL++ [19], as well as the state-of-the-art fuzzer of binary file formats AFLSmart [39]. We do not compare against the grammar-based fuzzers Superion [53], Nautilus [3] and Grimoire [5] because such fuzzers were only evaluated on text-based grammar input formats, such as markup languages (XML) or programming languages (JavaScript, PHP, Ruby, Lua, C, nasm, SQL, SMT). These textual formats do not have common features of binary formats, such as size fields, checksums, or bitfields. Therefore, it is not clear how well such fuzzers could handle those features. For Superion and Nautilus, a complete format specification would have to be written from scratch for each new binary format. On the other hand, Grimoire can mine format specifications, but it only supports textual languages, as it splits its inputs based on particular characters, such as opening and closing brackets and quotation marks. The Grimoire paper points out, for instance, that the tool can apply effective mutations to Lua source code, but not to Lua bytecode.

For evaluations, we have chosen 10 popular binary file formats, including the most popular formats for compressed archives (ZIP), images (PNG, JPG) and videos (MP4), as well as additional formats which are supported by AFLSmart.

7.1 RQ1: Effort for Extending Template Files

We start with RQ1: *How much effort is it to set up a binary template file for input generation?* To answer this question, Table 1 lists the number of lines of code required for each format specification. We list the sizes of the original binary template and modified version, which supports generation. We also detail the number of lines which were added or removed. For most formats the number of changes is small compared to the amount of code we can already leverage from existing parsing-only binary templates.

All of the formats shown in Table 1, with the exception of PNG, were developed by three different BSc students, who did not have prior experience with binary formats. Once they had learned the *binary template language* while creating the first format, development was easier, since many patterns are common to multiple formats. From their experience, a couple of days is sufficient for updating most formats for generation. However, a few complex formats took more than one week to modify. That was the case with MP4, which is composed of several different chunk types, many of which were not fully described in the original binary template. Note, though, that this effort can hardly be avoided: A random fuzzer will only very occasionally generate a valid MP4 file, let alone

Table 2. Average performance of the fuzzers in terms of speed and validity.

	Speed (files / *s*)		Size	With Evil		Without Evil		AFL	
Fmt.	Gen.	Parsed	(*B*)	Suc.	Valid	Suc.	Valid	Valid	Validation Command
PNG	6,229	2,488	752	97%	84%	100%	92%	5%	`identify -verbose - | grep Elapsed`
JPG	7,457	5,059	1,343	100%	81%	100%	95%	66%	`identify -verbose - | grep Elapsed`
GIF	6,111	3,760	6,007	100%	40%	100%	40%	14%	`identify -verbose - | grep Elapsed`
MIDI	3,453	21,639	573	96%	74%	100%	74%	60%	`! timidity - -Ol | grep ∧-:`
MP4	3,786	2,465	2,093	99%	82%	100%	88%	34%	`ffmpeg -y -i - -c:v mpeg4 -c:a copy o.mp4`
ZIP	10,229	13,844	1,894	96%	72%	98%	75%	2%	`yes | unzip -P "" -t` ⟨input file⟩[1]
PCAP	6,736	5,299	2,190	95%	91%	100%	100%	3%	`tcpdump -nr -`
AVI	18,489	25,560	415	95%	84%	100%	90%	27%	`ffmpeg -y -f avi -i - output.avi`
WAV	7,279	7,988	4,014	95%	96%	98%	99%	26%	`wavpack -y - -o output.wav`
BMP	4,377	6,225	1,173	95%	80%	95%	83%	23%	`identify -verbose - | grep Elapsed`

systematically cover all chunk types; and a hand-coded MP4 generator will require the same or higher specification effort (and neither result in a parser nor a mutator).

Table 1 also lists the number of lines of format-specific C++ code which were automatically generated by FormatFuzzer. We have found that the use of a domain-specific language of binary templates makes our specifications much more succinct and readable than if they were developed from scratch in a general-purpose language such as C++. Only for the PNG format did we have to edit the C++ code directly to support data compression; in the future, we plan to add native support for compression in FormatFuzzer, making such manual changes unnecessary.

To give some perpective, the first author of this paper had written a PNG generator completely from scratch in C++, before seeing the PNG binary template [45], by reading the PNG specification [14], a document which spans 50 pages. This manually written implementation had 596 lines of C++ code, which is a lot higher than the number of lines of code that had to be modified by starting from an existing PNG binary template and leveraging the FormatFuzzer framework—and it is a generator only, without any parsing or mutation capabilities, let alone coverage guidance.

Takeaway 1. *Extending binary templates is less work than writing a generator from scratch, and provides a parser and mutator on top.*

7.2 RQ2: Generator Speed

For RQ2 (*How* efficient *is FormatFuzzer in producing inputs?*), we measure the speed of our generators and parsers and classify the validity of the generated files by feeding them to a target program. The question of how to define the validity of files can be somewhat tricky. Different programs that process a given format tend not to agree about which inputs they accept, and they often do not follow the official specification. Programs such as media players tend to be forgiving and continue processing the input even after identifying some data corruption. In line with our main goal of reaching deeper code in the application, we classify a file as invalid only when it triggers a critical error which prevents further processing of the file. For example, for images we used the command `identify -verbose` from ImageMagick and checked the output to see if the program could successfully print detailed information about the image, even if some errors were reported.

Table 2 summarizes our results. For each format-aware fuzzer, we have generated 10,000 files, then also parsed those files and tested them with a target program to check for validity (validation command). Here, we see that FormatFuzzer can both generate and parse inputs at a speed of

[1]For `unzip`, both exit status 0 (normal) and exit status 1 (just warnings, but no errors) are considered valid files.

thousands per second for all the tested file formats. This is comparable or sometimes even faster than the speed with which format-agnostic fuzzers, such as AFL, can execute inputs. Therefore, our generators and parsers can be easily integrated into existing fuzzers without becoming a performance bottleneck.

Takeaway 2. *FormatFuzzer is very fast, generating and parsing thousands of inputs per second.*

7.3 RQ3: Input Validity

Table 2 also shows which fraction of the generations produced an output (successfully completed generation) and which fraction of those outputs was classified as valid, addressing RQ3: *How* accurate *is FormatFuzzer in producing inputs?* The results show that our fuzzers succeed almost all the time in producing inputs and those inputs have a high chance of being accepted by the target program. The validity is even higher if evil decisions are disabled. However, evil decisions are particularly useful in fuzzing to trigger some unexpected behaviors.

Takeaway 3. *The large majority of files generated by FormatFuzzer is valid.*

To put this validity into perspective, let us compare against inputs produced by the format-agnostic AFL fuzzer, listed under the "AFL" column. We see that even with sample files, AFL produces far fewer valid input files. In fact, if we compute the ratio of valid files produced by FormatFuzzer and valid files produced by AFL, we obtain geometric average of 4.9 (minimum 1.2, maximum 36.0).

Takeaway 4. *FormatFuzzer produces on average five times more valid inputs than a format-agnostic fuzzer.*

In the next section, we also compare the validity of inputs produced by FormatFuzzer against inputs produced by AFLSmart.

7.4 RQ4: Mutation Validity

For RQ4 (*How accurate are the smart mutations applied by FormatFuzzer?*), we evaluate how successful our smart mutations are compared to simpler mutations which do not use decision seeds. From our four smart mutations defined in Section 6.2, three of them (*replace*, *insert*, *delete*) could be implemented in a *simple* way by applying the corresponding operation (*replace*, *insert*, *delete*) directly to the chunks of the files, without using decision seeds. Such implementation is common in prior work, such as in AFLSmart [39].

In order to evaluate our smart mutations, we have applied each type of mutation 10,000 times over the initial corpus of valid files for each format. Table 3 show our results, where the numbers are expressed as percentages. For each smart mutation, we first show the success rate (which percentage of tries successfully generated a file). For the *replace* and *insert* mutations, we also show in parenthesis the percentage of tries for which we consumed exactly the expected number of decision bytes while generating the target chunk. In those cases, we should expect the resulting file to be the semantically correct result of applying the desired mutation. But even when this does not happen (for example, if the chunk to be inserted cannot fit in the target position), in the great majority of cases we still manage to produce a file, which is helpful for fuzzing. In fact, the success rate in generating a file is almost always above 91%.

From those cases where a file is successfully generated, we report which percentage of those files are considered valid, and also which percentage would be valid if we applied the *simple* version of the corresponding mutation. As shown in Table 3, we obtain almost universally higher validity when applying our smart mutations over decision seeds, when compared to their *simple* versions.

Table 3. Smart mutations: success (Suc.), semantic correctness of the target chunk (Cor.), validity (Val.), validity of the corresponding *simple* mutation (Sim.), and fraction of cases where our smart mutations differ from the *simple* versions (Diff.). All numbers are expressed as percentages (%).

	Replace				Insert				Delete				Abstract	
Fmt.	Suc. (Cor.)	Val.	Sim.	Diff.	Suc. (Cor.)	Val.	Sim.	Diff.	Suc.	Val.	Sim.	Diff.	Suc.	Val.
PNG	98 (36)	84	7	98	95 (12)	60	19	97	96	75	37	73	100	90
JPG	100 (58)	84	68	70	100 (38)	45	79	87	100	86	86	50	100	83
GIF	100 (61)	39	29	51	100 (60)	24	20	49	100	6	6	9	100	41
MIDI	100 (91)	99	92	45	100 (98)	100	99	100	–/–[2]	–/–	–/–	–/–	100	98
MP4	94 (39)	67	43	77	86 (16)	24	19	98	77	30	28	99	96	71
ZIP	99 (54)	77	27	81	96 (41)	71	29	100	99	77	0	100	99	83
PCAP	99 (86)	100	88	24	100 (100)	100	100	0	100	100	100	0	98	99
AVI	91 (53)	78	75	80	92 (37)	77	86	100	91	89	84	100	94	86
WAV	97 (45)	74	42	85	92 (27)	75	33	99	95	61	31	97	97	78
BMP	97 (58)	95	61	84	–/–[3]	–/–	–/–	–/–	–/–	–/–	–/–	–/–	99	93

This is expected, as our smart mutations are able to take contextual information into account, for example, being able to recompute the correct checksum for a chunk, even when the contents of the chunk have been partially modified. We also report on which percentage of the cases the files resulting from our smart mutations are different from what would be obtained by applying a *simple* mutation. Finally, we would also like to point out that our *smart abstract* mutations are only possible due to FormatFuzzer's ability to generate new chunks completely from scratch, and those abstractions produce very high validity, being the most profitable type of mutation in reaching new coverage during fuzzing.

Takeaway 5. *Smart mutations applied over decision seeds, as performed by FormatFuzzer, are far more likely to yield valid inputs than smart mutations which simply copy the contents of chunks from the files, as the ones used by prior work, such as AFLSmart.*

7.5 RQ5: Black-Box Fuzzing

Let us now evaluate the *effectiveness* of FormatFuzzer, notably in achieving *coverage* in our test subjects. Generally speaking, the higher the coverage of a test generator, the higher its chance to detect bugs—in particular, since if code is not covered, bugs will not be detected.

We start with RQ5: *How efficient is FormatFuzzer as a standalone black-box fuzzer?* We evaluate two black-box settings:

FFGen is using FormatFuzzer as a standalone input generator, requiring no knowledge or feedback from the program under test. (Note that this is a setting in which only a specification-based fuzzer like FormatFuzzer can succeed.)

FFMut is using FormatFuzzer to parse one initial set of input files and apply smart mutations, again without guidance.

First, we evaluate the *language coverage*, i.e., which features of the binary template are actually present in the generated files. In order to do this, we have generated 10,000 files with the black-box generation strategy FFGen and measured which percentage of variable declaration statements in the binary template have been covered, since variable declarations are the places where a new node is added to the parse tree. Table 5 shows the resulting language coverage, which is at least 94% for almost all formats. Only the JPG format had a lower coverage of 79%. The missing coverage was mostly due to some optional chunks which were enabled for parsing, but had not yet been

[2]The MIDI format does not have any deletable chunks.

[3]The BMP format does not have any optional chunks, so insertions and deletions are not possible.

Table 4. Programs under test.

Format	Program	Invocation
PNG	`libpng 1.6.37-3`	`readpng`
JPG	`libjpeg-turbo 1_2.0.6-4`	`djpeg` ⟨input file⟩
GIF	`gif2png 2.5.14`	`gif2png` ⟨input file⟩
MIDI	`TiMidity++ 2.14.0-8`	`timidity - -o - -Ow`
MP4	`FFmpeg 4.4`	`ffmpeg -y -i - out.mp4`
ZIP	`UnZip 6.0-26`	`unzip -P "" -t` ⟨input file⟩
PCAP	`tcpdump 4.99.0-2,` `libpcap 1.10.0-2`	`tcpdump -nr -`
AVI	`FFmpeg 4.4`	`ffmpeg -y -i - out.avi`
WAV	`WavPack 5.4.0-1`	`wavpack -y - -o -`
BMP	`libgdk-pixbuf 2.42.6-1`	`gdk-pixbuf-thumbnailer` ⟨input file⟩ `out.png`

Table 5. Language Coverage: coverage of variable declarations (%) in the binary template.

Setting	PNG	JPG	GIF	MIDI	MP4	ZIP	PCAP	AVI	WAV	BMP
FFGen	100	79	100	100	94	94	100	98	97	98

modified to generate valid contents. We generate such chunks only with very low probability, in order to ensure most generations will produce valid files.

Takeaway 6. *By itself, FormatFuzzer extensively covers the language features present in* valid *inputs.*

Next, we evaluate how well the fuzzer performs on real-world applications. Table 4 lists the programs with the exact command used for fuzzing. The fuzzing experiments used a timeout of 24 hours. For all techniques which require an initial corpus of inputs, we have used the same corpus of files, which were small files from each format manually downloaded from GitHub. Each corpus had an average of 11 files (minimum 5, maximum 31).

Table 6. Line coverage (%) of FormatFuzzer in black-box settings. Here, $A \setminus B$ represents the lines which were covered by technique A, but not B.

Setting	PNG	JPG	GIF	MIDI	MP4	ZIP	PCAP	AVI	WAV	BMP
FFGen	23.0	24.6	68.9	12.4	5.7	34.8	12.5	5.8	21.8	27.8
FFMut	24.7	24.4	70.5	10.4	7.4	36.1	9.8	7.1	21.9	27.8
FFGen \ FFMut	0.2	0.2	0.5	2.1	0.5	0.2	4.2	0.8	0.1	0.1
FFMut \ FFGen	1.9	0.1	2.2	0.1	2.2	1.5	1.5	2.0	0.2	0.0

Table 6 lists the resulting line coverage obtained with LCOV (average over all runs) for the black-box settings. One interesting aspect is that in both settings, all inputs are generated syntactically valid by construction; and indeed, inspection shows that error-handling code is hardly covered.[4]

Takeaway 7. *FormatFuzzer achieves decent coverage even in black-box settings, without any guidance from the program under test.*

[4]Generally speaking, reaching error-handling code is the easy part of fuzzing; if one wants to cover error-handling code, too, one could easily introduce subtle lexical mutations into the files generated by FormatFuzzer.

The lower half of Table 6 highlights the *coverage difference* between the two settings; here, $A \setminus B$ denotes lines covered in A, but not in B. The advantage of FFGen over FFMut is prominent in MIDI and PCAP, where the binary template covers exotic contents not found in our sample files. Conversely, the additional coverage of FFMut over FFGen is likely due to some features in the corpus of input files which are not described in the binary templates.

Takeaway 8. *Some features in the spec are unlikely to be found in files in the wild.*

Additionally, we have noticed that for all formats, the final coverage obtained after generating files with the FFGen approach exceeds the coverage from our downloaded corpus (by a factor of 1.4× on average).

7.6 RQ6: Integrating Format-Agnostic Fuzzers

In practice, we will frequently encounter settings where feedback from the program under test is available, and where FormatFuzzer therefore can integrate with existing format-agnostic fuzzers. Let us thus address RQ6: *How efficient is the integration of FormatFuzzer with a format-agnostic fuzzer?* We have integrated FormatFuzzer with the popular AFL++ [19] fuzzer in two different ways, as presented in Section 6.3:

AFL+FFGen is using AFL to mutate the *decision seed* which is then fed into FormatFuzzer and used to generate inputs for the target program. Since only the decision seed is mutated, the generated inputs should conform to the given format.

AFL+FFMut is having AFL use FormatFuzzer to perform smart mutations on the input file. FormatFuzzer parses the files which are about to be fuzzed and remembers information about the chunks present in those files, which it can later use to perform smart mutations. Note that all regular AFL mutations are also used, which can yield invalid inputs and helps covering error handling code. The smart mutations from FormatFuzzer are added as an additional custom mutator.

Table 7. Line coverage (%) of FormatFuzzer integrated with AFL. Here, $A \setminus B$ represents the lines which were covered by technique A, but not B. Vargha-Delaney effect size A^{12} (statistically significant effects in **bold**).

Setting	PNG	JPG	GIF	MIDI	MP4	ZIP	PCAP	AVI	WAV	BMP
AFL	20.7	29.8	73.3	13.3	11.5	36.9	25.3	11.2	22.4	30.7
AFL+FFGen	25.4	27.2	71.7	12.1	9.7	38.4	22.1	10.3	21.9	27.9
AFL+FFMut	27.8	33.6	73.3	14.3	11.4	38.6	26.2	11.8	22.3	30.7
AFL+FFGen \ AFL	6.3	2.4	0.1	0.3	0.6	2.4	1.1	0.7	0.0	0.0
AFL+FFMut \ AFL	7.2	3.8	0.0	1.7	0.9	2.1	2.3	1.0	0.1	0.0
AFL \ AFL+FFGen	1.5	5.0	1.8	1.5	2.5	0.9	4.3	1.6	0.5	2.9
AFL \ AFL+FFMut	0.0	0.0	0.1	0.7	0.9	0.4	1.5	0.5	0.3	0.0
AFL+FFGen \ FFGen	2.5	2.8	3.3	0.2	4.1	3.6	10.0	4.4	0.3	0.0
AFL+FFMut \ FFMut	3.2	9.2	2.9	3.9	4.1	2.6	16.5	4.5	0.4	2.9
A^{12}(AFL+FFMut > AFL)	**1.0**	**1.0**	0.34	0.66	0.52	**1.0**	**0.81**	**0.87**	0.41	0.45

Table 7 lists the resulting coverage. In our fuzzing experiments, we have enabled AFL dictionaries whenever suitable, for a fair comparison against the optimal setup of AFL. We have not used dictionaries for the AFL+FFGen approach, since this fuzzer mutates decision seeds, and not binary files directly. Dictionaries were enabled for all formats, with the exception of MIDI, for which there was no dictionary provided by AFL++.

For most subjects, the set differences show that both strategies increase coverage over plain AFL, showing that FormatFuzzer is effective in making format-agnostic fuzzers format aware. For six out of ten subjects, the integration also achieves a higher coverage in absolute terms.

Table 7 also reports the Vargha-Delaney effect size A^{12}, which estimates the probability that AFL+FFMut will perform better than the competing technique AFL. Values greater than 0.5 indicate that AFL+FFMut more likely to win this coverage comparison. Statistically significant effects are marked in **bold**, according to the Wilcoxon signed-rank test. As we can see, AFL+FFMut performed significantly better on five benchmarks, while for the remaining five there was no statistically significant difference. On three benchmarks, AFL had a slight advantage. The reason here was because the smart mutations from FormatFuzzer require slightly more computation time than format-agnostic mutations, so AFL achieved more coverage simply by being able to run more executions. But the advantage AFL had on those benchmarks is not statistically significant.

Takeaway 9. *Format-aware fuzzing with FormatFuzzer reaches lines that format-agnostic fuzzing does not.*

However, AFL by itself also covers lines that the integration of AFL and FormatFuzzer did not reach. One reason for this is again AFL producing several *invalid* inputs, covering error-handling code, which AFL+FFGen avoids by construction. The evil decisions in FormatFuzzer do enable a limited class of invalid inputs, but such evil decisions are sparse and are not allowed to completely destroy the structure of the input, for example, by generating inputs that fail to parse due to an incorrect chunk size. So the random mutations performed by AFL are still needed to cover all the parsing errors in the target program.

Takeaway 10. *Format-aware and format-agnostic fuzzing complement each other.*

Comparing the AFL integration against FormatFuzzer standalone, we also see that integrating AFL's coverage guidance into format-aware fuzzing also significantly improves coverage over a pure black-box setting. Hence, such integration is a preferred setting if sample inputs and coverage feedback are available.

Takeaway 11. *Integrating FormatFuzzer with coverage-guided fuzzers improves coverage over black-box settings.*

7.7 RQ7: Alternate Format-Aware Strategies

Our next evaluation concerns RQ7: *How does FormatFuzzer compare against other format-aware fuzzers?* The competitor here is AFLSmart [39], which uses format information from *Peach* specifications to determine chunk boundaries in input files, thus also resulting in smarter mutations. AFLSmart uses input specifications to *parse* inputs only, but not to *generate* valid inputs.

Table 8. Line coverage (%) of FormatFuzzer and AFLSmart. Here, $A \setminus B$ represents the lines which were covered by technique A, but not B. Vargha-Delaney effect size A^{12} (statistically significant effects in **bold**).

Setting	PNG	JPG	GIF	MIDI	MP4	ZIP	PCAP	AVI	WAV	BMP
AFLSmart	20.7	29.7	–/–[5]	14.1	12.3	36.4	26.4	12.2	22.2	–/–[6]
AFL+FFMut	27.8	33.6	73.3	14.3	11.4	38.6	26.2	11.8	22.3	30.7
AFLSmart $\setminus$ AFL+FFMut	0.0	0.1	–/–	1.7	1.3	0.2	1.9	1.0	0.0	–/–
AFL+FFMut $\setminus$ AFLSmart	7.1	4.0	–/–	1.9	0.5	2.4	1.7	0.6	0.1	–/–
A^{12}(AFL+FFMut > AFLSmart)	**1.0**	**1.0**	–/–	0.48	**0.04**	**1.0**	0.39	**0.12**	**0.73**	–/–

[5] AFLSmart crashed with a segmentation fault.

[6] BMP format is still not supported by AFLSmart.

The coverage results are listed in Table 8. Comparing AFLSmart against AFL+FFMut, we see that the AFL+FormatFuzzer integration outperforms AFLSmart in five out of eight subjects, while in the other two subjects (GIF and BMP) we could not run AFLSmart. Table 8 also includes the Vargha-Delaney effect sizes A^{12} and the Wilcoxon signed-rank test for statistical significance. We see that AFL+FFMut performed significantly better in four benchmarks, while AFLSmart had a significant advantage in two benchmarks. As we again see in the differences, even on cases where AFLSmart performed better, AFL+FFMut has still covered some lines of code which were not covered by AFLSmart.

One contributing factor here is the completeness of the format specifications. One advantage of FormatFuzzer is that we can leverage very detailed binary templates which had already been developed for parsing. The creators of AFLSmart, on the other hand, had to write their format specifications (*Peach pits*) from scratch, so they are less complete. For example, the *Peach pit* for PNG only specializes the internal contents of three chunk types: IHDR, cHRM and IEND, while the PNG binary template fully defined 15 chunk types, even before any changes.

However, from our experience, the more detailed format specifications lead to only small improvements in fuzzing if those specifications are only used for parsing, as in AFLSmart. FormatFuzzer reaps the most benefits from those detailed specifications from its ability to also generate valid inputs, which is used in FFMut to apply semantically valid smart mutations which respect contextual information.

Takeaway 12. *Using input formats for parsing, mutating, and generating, as FormatFuzzer does, yields additional coverage over using them for parsing only, as AFLSmart does.*

We have also noticed that FormatFuzzer is particularly efficient in generating high-quality inputs and achieving high coverage very quickly during the fuzzing campaign.

Table 9. Line coverage (%) of FormatFuzzer and AFLSmart in **one hour** of fuzzing. Here, $A \setminus B$ represents the lines which were covered by technique A, but not B. Vargha-Delaney effect size A^{12} (statistically significant effects in **bold**).

Setting	PNG	JPG	GIF	MIDI	MP4	ZIP	PCAP	AVI	WAV	BMP
AFLSmart	20.5	28.5	−/−	10.4	8.3	33.9	17.1	8.0	22.1	−/−
AFL+FFMut	27.3	31.4	71.9	10.4	8.5	37.2	18.1	8.2	22.2	30.0
AFLSmart $\setminus$ AFL+FFMut	0.0	0.4	−/−	0.1	0.4	0.2	1.0	0.3	0.0	−/−
AFL+FFMut $\setminus$ AFLSmart	6.8	3.3	−/−	0.0	0.6	3.4	2.0	0.6	0.1	−/−
A^{12}(AFL+FFMut > AFLSmart)	**1.0**	**1.0**	−/−	0.43	0.73	**1.0**	**0.92**	**0.86**	**0.97**	−/−

Table 9 shows the same information as in Table 8, but only for the first hour of fuzzing (we use again an average over 10 runs). We notice that, in the short run of one hour, FormatFuzzer performed significantly better than AFLSmart on six out of eight benchmarks, and never performed significantly worse.

Takeaway 13. *FormatFuzzer is particularly good at finding coverage on a limited time budget.*

7.8 RQ8: Bugs Found

We close our evaluation with RQ8: *Does FormatFuzzer find real bugs?* The answer is: Yes! In our initial fuzzing experiments with FormatFuzzer, we have found and reported:

- 16 distinct segmentation faults (by different stack traces) and 8 distinct aborts in `ffmpeg` (MP4 and AVI), which turned out to be at least 8 distinct bugs which have already been fixed to date by the FFmpeg developers. These occur when `ffmpeg` is compiled with pthreads

(which is the default option) and the fuzzer runs with a memory limit of 200 MB (using AFL's `-m` option). These bugs are located in a wide range of different stages through the execution: allocation, initialization of h264 slices, multi-threading, writing `mov` packets, video encoding, decoding spectral data and context cleanup.
- 19 distinct memory errors (by different stack traces) in `timidity` (MIDI). We will update the paper when those bugs are fixed.

Note that `ffmpeg` "is part of the workflow of hundreds of other software projects, and its libraries are a core part of software media players such as VLC, and has been included in core processing for YouTube and iTunes." [1]

Takeaway 14. *FormatFuzzer finds bugs in relevant software.*

7.9 Discussion

Leveraging input format specifications pays off. In every single setting, FormatFuzzer explored additional coverage—and thus additional chances to find bugs and vulnerabilities. FormatFuzzer can be used as a standalone generator, notably in black-box settings; the integration with feedback-driven fuzzers such as AFL combines the best of format-aware and feedback-directed fuzzing.

The evaluation also shows the *versatility* of FormatFuzzer. Its individual components (parsing/-mutating/generating) can either be used standalone (say, *generating* in a black-box setting), or fully integrated with format-agnostic fuzzers (where the popular AFL tool can be replaced by any even more performant format-agnostic fuzzer).

This versatility and modularity makes FormatFuzzer a *platform* for quickly creating and integrating novel fuzzing strategies and adapting them towards the setting at hand—in contrast to fuzzers which have no knowledge about the input structure, or which use input specifications exclusively for parsing or for generating. As our evaluation shows, both format-aware and format-agnostic strategies have their specific strengths, further motivating the need for an integrative platform.

Another important aspect not addressed previously is that, by construction, FormatFuzzer gives the tester *control* over what should be tested. By commenting out particular parts of a binary template, FormatFuzzer allows to focus on specific features—say, those that were recently changed or otherwise are critical. This adds to the versatility of fuzzing with FormatFuzzer.

8 RELATED WORK

8.1 Fuzzing with Input Specifications

Using language specifications for generating inputs is an old idea, especially for context-free grammars. The different grammars such as regular, context-free, context-sensitive, and unconstrained grammars were invented by Chomsky for linguistic applications [9] in 1950s specifically for parsing. Using grammars for input generation was first suggested by Burkhardt [6], Hanford [26] and Purdom [40] in the late 1960s and early 1970s. Logic languages [24] such as Prolog [55] which are known from 1970s allow turning the entire program inside out, using the program itself to generate the kind of inputs that it accepts. Free generators [22] are a recent formalism which also unifies parsing with generation, as in FormatFuzzer. However, they focus on the generation of data structures, such as type-classes in Haskell, while FormatFuzzer targets binary formats and integrates with existing format-agnostic fuzzers.

8.2 Context-Free Grammars

Grammar-based testing took off in the new millennium when researchers rediscovered the utility of fuzzing, and how grammars can improve the effectiveness of fuzzing. One of the first to recognize their use in fuzzing was Godefroid [21] who augmented whitebox fuzzing with grammars. Other noteworthy grammar fuzzers include Gramfuzz [25], Grammarinator [28], Dharma [34], Domato [20], and CSS Fuzz [41], as well as PolyGlot [7], which augments context-free grammars with semantic annotations. LangFuzz [29] uses a language specification to collect code fragments which can be applied as smart mutations. All these work on context-free languages, which are not sufficient to specify binary formats.

8.3 Other Specification Languages

Besides context-free grammars, fuzzers have used alternate input specifications, such as regular languages (i.e. finite state automata) [11, 54], or constraint languages [15]. SISL [50] uses a partial specification which is complemented by the use of concolic testing [42]. MoWF [38] leverages a specification expressed as a constraint over the input space. It uses selective symbolic execution to identify uncovered branches, and can repair length fields, checksums and other validation fields. The fuzzer by Pan et al. [37] leverages higher order attribute grammars to describe length, checksums, and other validation fields, and uses them to generate file format inputs such as PNGs. Parsifal [32] targets both parsing and generation of binary formats. It is, however, limited to fixed size formats. Nail [4] defines a parser generator specification, and targets binary formats (typically protocols) that contain offset fields and checksums, and unlike Parsifal, can handle more complex constructs. Underwood [51] extends context-free grammars using attribute grammars, and provides a mapping from binary format specifications to attribute grammars. Such grammars can be used for parsing as well as generation. Beginner's luck [30] is a language for generators that allows the integration of sampling constraints to be used during fuzzing. libprotobuf-mutator [44] uses protocol buffers to describe file formats, allowing the mutations to be done over the compact protocol buffer representation. However, one disadvantage compared to FormatFuzzer is the need to write a converter between the protocol buffer and the corresponding binary file.

In principle, FormatFuzzer could make use of any of these format specifications, and still leverage its unique capabilities such as generating synchronized parsers and generators, or integrating format-agnostic fuzzers. By using binary templates, however, FormatFuzzer can build on hundreds of format specifications that have been created and refined by an enthusiastic community for more than two decades now.

8.4 Fuzzing Strategies

CSmith [57] shows that embedding the entire language specification into the generator can lead to effective fuzzing. However, with that, one loses the generality of being able to target other kinds of inputs. Other notable research on grammar-based fuzzers include LangFuzz [29], Blendfuzz [56], Skyfire [52]. *Parameter sequences* in Zest [36] and JQF [35] encode generator choices like our *decision seeds* and can be used for generator-based testing in the style of QuickCheck [10]. Crowbar [16] and CGPT [31] similarly also combine generator-based testing with coverage feedback, allowing both generation-based and mutation-based fuzzing. However, obtaining the decision seeds by parsing existing files, as well as applying smart mutations on them (Section 6.2) is a novel contribution of FormatFuzzer.

Several grammar-based fuzzers also incorporate coverage feedback from AFL. Notable examples include Superion [53], Nautilus [3] and Grimoire [5]. However, those fuzzers were only applied to text-based grammar input formats, such as markup languages (XML) or programming languages

(JavaScript, PHP, Ruby, Lua, C, nasm, SQL, SMT). So they would likely be unable to support features required for binary file formats, such as size fields, checksums, or bitfields. Superion and Nautilus would require a complete format specification to be written from scratch for each new format. Grimoire, on the other hand, can mine format specifications, but only for textual languages.

8.5 Fuzzing Binary Formats

WEIZZ [18] and FFAFuzz [8] are fuzzers which target chunk-based binary formats. During fuzzing, they try to learn how the chunk-based format is specified. Peach [17] is another fuzzer that targets binary formats. AFLSmart [39] is a fuzzer that provides chunk aware mutations to the input, and hence targets binary formats. It does this by maintaining a virtual structure of the input being fuzzed in memory. AFLSmart is the closest related work to FormatFuzzer. However, since its format specifications (*Peach pits*) had to be developed from scratch for use with AFLSmart, they are less complete and comprehensive than our binary templates. Since FormatFuzzer can use its binary templates not only for parsing, but also for generation, this enables additional fuzzing strategies detailed in this paper.

9 CONCLUSION

In order to fuzz binary inputs, the most powerful solution is to create a dedicated fuzzer. However, creating such fuzzers takes significant effort. With FormatFuzzer, one can leverage *hundreds of existing binary templates* to make existing fuzzers format-aware. FormatFuzzer can readily use such binary templates as *parsers,* which reveal the full structure of inputs. Extending existing binary templates for generation unlocks fuzzing in black-box settings that are infeasible for format-agnostic fuzzers. By providing decision seeds and smart mutations, FormatFuzzer can make any fuzzer format-aware.

While FormatFuzzer is a highly efficient choice for domain-specific fuzzing, its modularity also makes it a useful platform for creating future fuzzers. Besides extending FormatFuzzer with further formats, our future work will focus on the following topics:

Black-box strategies. Recent advances in grammar-based fuzzing, such as systematically achieving grammar coverage [27] or learning and leveraging probability distributions [49] could easily be adopted for binary template fuzzing, too.

Search-based fuzzing. The *decision seed* format opens way for easily integrating alternate fuzzing strategies. Of particular interest is *genetic optimization* as part of *search-based testing,* since FormatFuzzer already implements mutation and crossover operations.

Mining binary formats. Recent techniques for mining context-free grammars from parsers [23, 33] could be adapted to binary formats, simplifying template construction.

Mining binary constraints. Extending a (mined) syntactic template, we can track processing of individual input elements using dynamic tainting and dynamic analysis, and extract context-sensitive constraints from input processors.

Engaging the community. We will be creating tutorials and other material to engage the community in writing binary templates. The Wikipedia list of file formats [2] lists more than 1,000 formats—so there is still lots to do!

FormatFuzzer and all supporting material is available as open source. For more information on FormatFuzzer, see its project page

https://uds-se.github.io/FormatFuzzer/.

ACKNOWLEDGEMENTS

We thank Pit Jost, Andreas Hanuja and Steven Dlucik for their work in supporting binary templates for multiple formats.

REFERENCES

[1] 2021. Wikipedia: ffmpeg. https://en.wikipedia.org/wiki/FFmpeg. Accessed May 2021.

[2] 2021. Wikipedia: List of File Formats. https://en.wikipedia.org/wiki/List_of_file_formats. Accessed May 2021.

[3] Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, Ahmad-Reza Sadeghi, and Daniel Teuchert. 2019. NAUTILUS: Fishing for Deep Bugs with Grammars. In *Proceedings of NDSS 2019*. https://www.ndss-symposium.org/ndss-paper/nautilus-fishing-for-deep-bugs-with-grammars/

[4] Julian Bangert and Nickolai Zeldovich. 2014. Nail: A practical tool for parsing and generating data formats. In *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*. 615–628.

[5] Tim Blazytko, Matt Bishop, Cornelius Aschermann, Justin Cappos, Moritz Schlögel, Nadia Korshun, Ali Abbasi, Marco Schweighauser, Sebastian Schinzel, Sergej Schumilo, et al. 2019. {GRIMOIRE}: Synthesizing structure while fuzzing. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*. 1985–2002.

[6] W. H. Burkhardt. 1967. Generating test programs from syntax. *Computing* 2, 1 (March 1967), 53–73. https://doi.org/10.1007/BF02235512

[7] Yongheng Chen, Rui Zhong, Hong Hu, Hangfan Zhang, Yupeng Yang, Dinghao Wu, and Wenke Lee. 2021. One Engine to Fuzz 'em All: Generic Language Processor Testing with Semantic Validation (to appear). In *Proceedings of the 42nd IEEE Symposium on Security and Privacy (Oakland)*. San Francisco, CA.

[8] Zehan Chen, Yuliang Lu, Kailong Zhu, Lu Yu, and Jiazhen Zhao. 2022. Fast Format-Aware Fuzzing for Structured Input Applications. *Applied Sciences* 12, 18 (2022), 9350.

[9] Noam Chomsky. 1956. Three models for the description of language. *IRE Transactions on Information Theory* 2 (1956), 113–124. https://chomsky.info/wp-content/uploads/195609-.pdf

[10] Koen Claessen and John Hughes. 2011. QuickCheck: a lightweight tool for random testing of Haskell programs. *ACM SIGPLAN Notices* 46, 4 (2011), 53–64.

[11] Baojiang Cui, Shurui Liang, Shilei Chen, Bing Zhao, and Xiaobing Liang. 2014. A novel fuzzing method for Zigbee based on finite state machine. *International Journal of Distributed Sensor Networks* 10, 1 (2014), 762891.

[12] James "d0c_s4vage" Johnson. 2020. GitHub - d0c-s4vage/pfp: pfp - Python Format Parser - a python-based 010 Editor template interpreter. https://github.com/d0c-s4vage/pfp. Accessed August 1, 2021.

[13] James "d0c_s4vage" Johnson. 2020. GitHub - d0c-s4vage/py010parser: A modified pycparser to parse 010 templates. https://github.com/d0c-s4vage/py010parser. Accessed August 1, 2021.

[14] Oxford Brookes University (Second Edition) David Duce. 2003. Portable Network Graphics (PNG) Specification (Second Edition). https://www.w3.org/TR/PNG/. Accessed November 15, 2021.

[15] Kyle Dewey, Jared Roesch, and Ben Hardekopf. 2014. Language fuzzing using constraint logic programming. ACM, 725–730.

[16] Stephen Dolan. 2021. Crowbar. https://github.com/stedolan/crowbar. Accessed November 15, 2021.

[17] Michael Eddington. 2011. Peach fuzzing platform. *Peach Fuzzer* 34 (2011).

[18] 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.

[19] 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)*. USENIX Association.

[20] Ivan Fratric. 2019. *Domato A DOM fuzzer*. https://github.com/googleprojectzero/domato

[21] Patrice Godefroid, Adam Kiezun, and Michael Y. Levin. 2008. Grammar-based Whitebox Fuzzing. ACM, New York, NY, USA, 206–215.

[22] Harrison Goldstein and Benjamin C. Pierce. 2022. Parsing Randomness. *Proc. ACM Program. Lang.* OOPSLA (2022).

[23] Rahul Gopinath, Björn Mathis, and Andreas Zeller. 2020. 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*. 172–183.

[24] Claude Cordell Green. 1970. *The application of theorem proving to question-answering systems*. Number 96. Management Information Services.

[25] Tao Guo, Puhan Zhang, Xin Wang, and Qiang Wei. 2013. Gramfuzz: Fuzzing testing of web browsers based on grammar analysis and structural mutation. In *2013 Second International Conference on Informatics & Applications (ICIA)*. IEEE, 212–215.

[26] Kenneth V. Hanford. 1970. Automatic Generation of Test Cases. *IBM Syst. J.* 9, 4 (Dec. 1970), 242–257. https://doi.org/10.1147/sj.94.0242

[27] Nikolas Havrikov and Andreas Zeller. 2019. Systematically Covering Input Structure. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering* (San Diego, California) *(ASE '19)*. IEEE Press, 189–199. https://doi.org/10.1109/ASE.2019.00027

[28] Renáta Hodován, Ákos Kiss, and Tibor Gyimóthy. 2018. Grammarinator: a grammar-based open source fuzzer. In *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation.* ACM, 45–48.

[29] Christian Holler, Kim Herzig, and Andreas Zeller. 2012. Fuzzing with Code Fragments. In *Proceedings of the 21st USENIX Conference on Security Symposium* (Bellevue, WA) *(Security'12)*. USENIX Association, Berkeley, CA, USA, 38–38.

[30] Leonidas Lampropoulos, Diane Gallois-Wong, Cătălin Hriţcu, John Hughes, Benjamin C Pierce, and Li-yao Xia. 2017. Beginner's luck: a language for property-based generators. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages.* 114–129.

[31] Leonidas Lampropoulos, Michael Hicks, and Benjamin C Pierce. 2019. Coverage guided, property based testing. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 1–29.

[32] Olivier Levillain. 2014. Parsifal: A pragmatic solution to the binary parsing problems. In *2014 IEEE Security and Privacy Workshops*. IEEE, 191–197.

[33] Björn Mathis, Rahul Gopinath, Michaël Mera, Alexander Kampmann, Matthias Höschele, and Andreas Zeller. 2019. Parser-directed fuzzing. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation.* 548–560.

[34] Mozilla. 2019. *Dharma: A generation-based, context-free grammar fuzzer.* https://blog.mozilla.org/security/2015/06/29/dharma/

[35] Rohan Padhye, Caroline Lemieux, and Koushik Sen. 2019. JQF: coverage-guided property-based testing in Java. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis.* 398–401.

[36] Rohan Padhye, Caroline Lemieux, Koushik Sen, Mike Papadakis, and Yves Le Traon. 2019. Semantic fuzzing with zest. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis.* 329–340.

[37] Fan Pan, Ying Hou, Zheng Hong, Lifa Wu, and Haiguang Lai. 2013. Efficient Model-based Fuzz Testing Using Higher-order Attribute Grammars. *JSW* 8, 3 (2013), 645–651.

[38] Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. 2016. Model-based whitebox fuzzing for program binaries. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering.* 543–553.

[39] Van-Thuan Pham, Marcel Böhme, Andrew Edward Santosa, Alexandru Razvan Caciulescu, and Abhik Roychoudhury. 2019. Smart greybox fuzzing. *IEEE Transactions on Software Engineering* (2019).

[40] Paul Purdom. 1972. A sentence generator for testing parsers. *BIT Numerical Mathematics* 12, 3 (1972), 366–375. https://doi.org/10.1007/BF01932308

[41] Jesse Ruderman. 2007. *Introducing jsfunfuzz.* http://www.squarefree.com/2007/08/02/introducing-jsfunfuzz/

[42] Koushik Sen, Darko Marinov, and Gul Agha. 2005. CUTE: A Concolic Unit Testing Engine for C. In *ESEC/FSE'05.*

[43] Kosta Serebryany. 2016. Continuous fuzzing with libfuzzer and addresssanitizer. In *2016 IEEE Cybersecurity Development (SecDev)*. IEEE, 157–157.

[44] Kostya Serebryany, Vitaly Buka, and Matt Morehouse. 2017. Structure-aware fuzzing for Clang and LLVM with libprotobuf-mutator. (2017).

[45] SweetScape Software. 2021. 010 Editor - Binary Template Repository - Download Binary Templates. https://www.sweetscape.com/010editor/repository/templates/. Accessed August 1, 2021.

[46] SweetScape Software. 2021. 010 Editor - Binary Templates - Parsing Binary Files. https://www.sweetscape.com/010editor/templates.html. Accessed August 1, 2021.

[47] SweetScape Software. 2021. 010 Editor - Pro Text/Hex Editor | Edit 160+ Formats | Fast & Powerful. https://www.sweetscape.com/010editor/. Accessed August 1, 2021.

[48] SweetScape Software. 2021. 010 Editor Manual - Writing Templates. https://www.sweetscape.com/010editor/manual/IntroTemplates.htm. Accessed August 1, 2021.

[49] Ezekiel Soremekun, Esteban Pavese, Nikolas Havrikov, Lars Grunske, and Andreas Zeller. 2020. Inputs from Hell: Learning Input Distributions for Grammar-Based Test Generation. *IEEE Transactions on Software Engineering* (2020), 1–1. https://doi.org/10.1109/TSE.2020.3013716

[50] Sören Tempel, Vladimir Herdt, and Rolf Drechsler. 2022. SISL: Concolic Testing of Structured Binary Input Formats via Partial Specification. In *Automated Technology for Verification and Analysis: 20th International Symposium, ATVA 2022, Virtual Event, October 25–28, 2022, Proceedings.* Springer, 77–82.

[51] William Underwood. 2012. Grammar-Based Specification and Parsing of Binary File Formats. *International Journal of Digital Curation* 7 (03 2012), 95–106. https://doi.org/10.2218/ijdc.v7i1.217

[52] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2017. Skyfire: Data-driven seed generation for fuzzing. IEEE, 579–594.

[53] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2019. Superion: grammar-aware greybox fuzzing. In *Proceedings of the 41st International Conference on Software Engineering*. IEEE Press, 724–735.

[54] Ming-Hung Wang, Han-Chi Wang, You-Ru Chen, and Chin-Laung Lei. 2017. Automatic Test Pattern Generator for Fuzzing Based on Finite State Machine. *Security and Communication Networks* 2017 (2017).

[55] David HD Warren, Luis M Pereira, and Fernando Pereira. 1977. Prolog-the language and its implementation compared with Lisp. *ACM SIGPLAN Notices* 12, 8 (1977), 109–115.

[56] Jingbo Yan, Yuqing Zhang, and Dingning Yang. 2013. Structurized grammar-based fuzz testing for programs with highly structured inputs. *Security and Communication Networks* 6, 11 (2013), 1319–1330.

[57] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in C compilers. In *ACM SIGPLAN Notices*, Vol. 46. ACM, 283–294.

[58] Michał Zalewski. 2016. American Fuzzy Lop. http://lcamtuf.coredump.cx/afl. Accessed October 1, 2016.