Model-Based Whitebox Fuzzing for Program Binaries

Van-Thuan Pham Marcel Böhme Abhik Roychoudhury
School of Computing, National University of Singapore, Singapore
{thuanpv, marcel, abhik}@comp.nus.edu.sg

ABSTRACT

Many real-world programs take highly structured and complex files as inputs. The automated testing of such programs is non-trivial. If the test does not adhere to a specific file format, the program returns a parser error. For symbolic execution-based whitebox fuzzing the corresponding error handling code becomes a significant time sink. Too much time is spent in the parser exploring too many paths leading to trivial parser errors. Naturally, the time is better spent exploring the functional part of the program where failure with valid input exposes deep and real bugs in the program.

In this paper, we suggest to leverage information about the file format and data chunks of existing, valid files to swiftly carry the exploration beyond the parser code. We call our approach Model-based Whitebox Fuzzing (MoWF) because the file format input model of blackbox fuzzers can be exploited as a constraint on the vast input space to rule out most invalid inputs during path exploration in symbolic execution. We evaluate on 13 vulnerabilities in 8 large program binaries with 6 separate file formats and found that MoWF exposes all vulnerabilities while both, traditional whitebox fuzzing and model-based blackbox fuzzing, expose only less than half, respectively. Our experiments also demonstrate that MoWF exposes 70% vulnerabilities without any seed inputs.

CCS Concepts:
•**Software and its engineering** → **Software testing and debugging;** •**Security and privacy** → *Vulnerability scanners;*

Keywords: Symbolic Execution, Program Binaries

1. INTRODUCTION

Testing file-processing programs can be challenging. Even though a structured file is stored as a vector of input bytes, it is often parsed as a tree where data chunks contain fields and other data chunks.

Our key insight is that certain branches in a file-processing program are exercised only depending on i) the *presence* of a specific data chunk, ii) a *specific value* of a data field in a data chunk, or iii) the *integrity* of the data chunks. Hence, an efficient test generation technique not only sets specific values of the fields but also adds/removes complete chunks and establishes their integrity (e.g., checksum or size).

ASE'16, September 3–7, 2016, Singapore, Singapore
© 2016 ACM. 978-1-4503-3845-5/16/09...
http://dx.doi.org/10.1145/2970276.2970316

Fuzzers help to test such file-processing programs. Model-based blackbox fuzzers (MoBF) [3, 5] utilize input models to generate *valid* random files. The input model specifies the format of the data chunks and integrity constraints. However, while valid, the modification is still inherently random. Whitebox fuzzers (WF) employ symbolic execution to explore program paths more systematically. Given a valid file, they can generate the specific values for the data *fields* quite comfortably. However, when it comes to adding or deleting data *chunks* or enforcing integrity constraints, they are bogged down by the large search space of invalid inputs [27].

Grammar-based whitebox fuzzers (GWF) can generate files that are valid w.r.t. a context-free grammar [14]. Like WF, GWF computes path constraints: logical formulas that are satisfied only by new files exercising *alternative* paths. Unlike WF, these constraints are converted into regular expressions such that a context-free constraint solver can generate an input that is accepted by both, the grammar and the expression. However, the expression is much weaker than the path constraint. Suppose, symbolic execution yields the path constraint $\varphi \wedge (x < y)$. After conversion, the regular expression cannot capture that arithmetic constraint. Moreover, GWF cannot encode integrity constraints such as size-of, offset-of, length-of and checksums. These integrity checks are very common in several highly structured file formats like PNG, PDF and WAV.

In this work, we present *Model-based Whitebox Fuzzing* (MoWF), an automated testing technique for industrial-size program binaries that process structured inputs. MoWF is a marriage of model-based blackbox fuzzing and whitebox fuzzing that generates valid files efficiently that exercise critical target locations effectively. It is a *directed path exploration* technique that prunes from the search space those paths that are exercised by invalid, malformed inputs: (i) MoWF uses information about the file format to explore those branches that are exercised depending on the presence of specific chunks. To this end, MoWF removes the referenced chunk or adds a new valid chunk by instantiation from the input model or a process we call *data chunk transplantation* — MoWF identifies the set of input bytes corresponding to the required chunk in a donor file and transplants them into the appropriate location of the receiving file. (ii) MoWF employs selective symbolic execution [11] to explore those branches that are exercised depending on specific values of the data fields. (iii) Lastly, MoWF establishes the integrity of the generated files, repairing checksums and offsets.

Unlike MoBF, MoWF is directed and enumerates the specific values of data fields more systematically. Unlike WF, MoWF does not get bogged down by the large search space of invalid inputs or require any seed inputs (cf. [13, 16]). Unlike GWF, MoWF maintains *full* path constraints so it has no impact on the soundness and completeness of WF. Moreover, MoWF leverages a more expressive yet simple input model to handle integrity constraints.

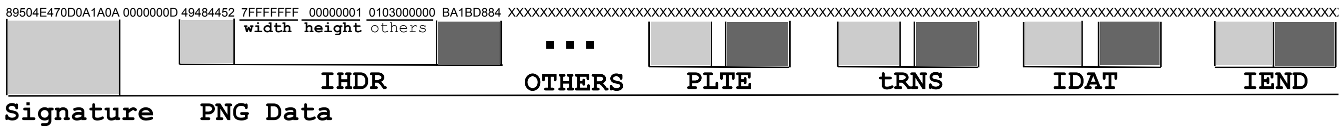

Figure 1: The structure and the hex code of a PNG file. A data chunk is a section in the hex code embedding one piece of information about the image. The hex code above the light-grey boxes identifies the data chunk type while the hex code above the dark-grey boxes protects the correctness of the data chunk (via checksum).

The input model is used to generate valid files efficiently, enforce integrity constraints, and facilitate the transplantation of data chunks. Since it only prunes search space, the input model does not need to be complete. On one hand, whitebox fuzzing eventually constructs all relevant (semi-) valid files by exploring paths that are not pruned by the input model. On the other hand, transplantating data chunks from donors maintains underspecified integrity constraints, such as the concrete compression algorithm with which the image data in a PNG file must be encoded. An input model is constructed once and can be used across all future testing sessions. It has been shown that input models can also be derived in an automated fashion [19, 18, 17]. Each of our input models was constructed manually in less than a day.

The two *main challenges* of Traditional Whitebox Fuzzing (TWF) that we address are:

- **Path Explosion**. Parser code is often a large and very complex part of a program. In practice, TWF gets bogged down by an exponential number of paths in the parser that are exercised by invalid inputs [27].
- **Seed Dependendence**. Most TWF approaches assume the existence of a seed file that features all necessary data chunks – it is only a matter of setting the correct values for the data fields to expose an error. In practice, however, this may not be the case. Data chunks may be missing or in the wrong order. In other cases no seed files may be available at all.

The main contributions of MoWF are as follows.

- **Pruning Invalid Paths**. The input model allows to prune most paths that are exercised by invalid inputs. As opposed to TWF, MoWF is capable of negating those *crucial* branches that are exercised only in the presence of certain data chunks without having to iteratively construct the data chunk by exploring the parser code. All generated test inputs are valid in that they adhere to the input model. Integrity constraints are enforced. Given a 24h time budget, our MoWF tool exposed all of thirteen vulnerabilities in our experimental subjects while the TWF tool exposed only six.
- **Reduced Seed Dependence**. The instantiation from the input model allows to construct seed inputs from scratch. Moreover, given a seed input that is missing a data chunk to reach a target location, MoWF allows to utilize other seed files as donors, transplant the missing data chunk, and construct a new seed input that is closer to the target location. In the absence of a donor, the missing data chunk can be directly instantiated from the input model. Out of the thirteen vulnerabilities in our experimental subjects our MoWF tool exposed nine *without any seed inputs*.
- **Fuzzing tool**. We implement our MoWF tool as an extension of the TWF tool, HERCULES [22]. We compare our MoWF tool not only to the HERCULES TWF but also to the PEACH model-based blackbox fuzzer [3]. Given a 24h time budget, our MoWF tool exposed all of 13 vulnerabilities in our experimental subjects while the both HERCULES and PEACH tool exposed only six.

Insights. Through our experiments we also gain insights about the relative strengths of our technique MoWF, symbolic execution based traditional whitebox fuzzing (TWF), and model-based blackbox fuzzing (MoBF) as in fuzzers like Peach/Spike [3, 5]. TWF performs well only if there exists a seed input that features all necessary data chunks and only certain values for data fields need to be set. MoBF performs well if the vulnerability is exposed by putting boundary values for certain data fields, or by removing/adding empty data chunks. Deep vulnerabilities that require specific values are best exposed by a symbolic execution-based approach. MoWF performs well even in the absence of seed inputs and swiftly generates the specific values needed to expose even deep vulnerabilities, while also gaining the capability to add and remove complete data chunks as in MoBF.

2. OVERVIEW

2.1 Motivating Example

We motivate MoWF based on a real, serious vulnerability in a library that is shipped with several browsers and media players. LibPNG [24] is the official PNG reference library; it supports almost all PNG features, and has been extensively tested for over 20 years. The library is integrated into popular programs such as VLC media player, Google Chrome web brower and Apple TV.

PNGs consist of four mandatory and fourteen optional types of data chunks. For easy parsing and error detection the file format requires to specify the size, type, and checksum of each data chunk besides the actual data. The particular PNG file in Figure 1 happens to expose a memory access violation vulnerability (OSVDB-95632) in VLC 2.0.7 [25] which uses LibPNG 1.5.14. To trigger the bug, the image width defined in the IHDR chunk must take a specific value (from `0x7FFFFFF2` to `0x7FFFFFFF`) and the optional tRNS chunk must exist. The tRNS chunk specifies alpha values to control the transparency of pixels in the image.

Figure 1 partially shows structure of a file that exposes the bug. The first eight bytes identify the file as PNG. The next four bytes specify the *size* of the next data chunk (`0xD` $=$ hex(13) bytes), followed by four bytes identifying the *type* of the chunk as IHDR (light-grey box). The next 13 bytes are data fields specifying image width and height. This is followed by four bytes of *checksum* protecting the correctness of the IHDR chunk (dark-grey box). The remaining chunks are structured similarly. The image data in the IDAT chunk is compressed using the DEFLATE compression algorithm [1] and the end of the PNG file is indicated by IEND chunk.

Listing 1 shows the pertinent code in LibPNG. In each iteration, `png_read_info` (lines 2-27) parses information about the current chunk, like its size and type. Depending on the type it calls the corresponding function to handle the current chunk and validate the checksum. These handler functions parse a chunk's data fields and store their values for further image transformation and processing steps. The chunks are parsed until the first IDAT chunk is reached (lines 18-22). The file shown in Figure 1 passes all checks in the parser and chunk-handling code and is therefore *valid*.

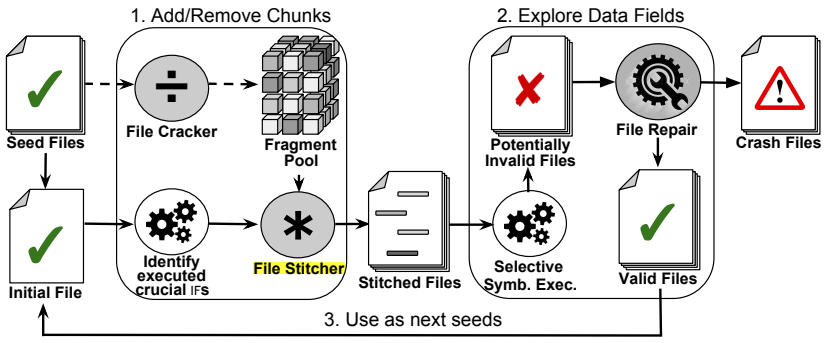

Figure 2: Model-based Whitebox Fuzzing. Elements marked in grey are informed by the data model.

```
1  // read chunks' info before first IDAT chunk
2  void png_read_info(png_structp ptr)
3  {
4  // read and check the PNG file signature
5  read_sig(f);
6  for (;;)
7  {
8    // get current chunk's information
9    uint_32 length = read_chunk_header(ptr);
10   uint_32 chunk_name = ptr->chunk_name;
11   // mandatory chunks
12   if (chunk_name == png_IHDR)
13      handle_IHDR(ptr, length);
14   else if (chunk_name == png_IEND)
15      handle_IEND(ptr, length);
16   else if (chunk_name == png_PLTE)
17      handle_PLTE(ptr, length);
18   else if (chunk_name == png_IDAT)
19   {
20      ptr->idat_size = length;
21      break;
22   }
23   // optional chunks
24   else if ...
25   else if (chunk_name == png_tRNS)
26      handle_tRNS(ptr, length);
27   else if ...
28 }
29 }
30 // initialize row buffer for reading data from file
31 void png_read_start_row(png_structp ptr)
32 {
33    size_t buf_size;
34    ...
35    buf_size = calculateBufSize(ptr);
36    ptr->row_buf = png_malloc(ptr, buf_size);
37    png_memset(ptr->row_buf, 0, ptr->rowbytes);
38 }
```

Listing 1: Simplified parser code for data chunks. The code is shown to ease the explanation; MoWF works directly with program binaries.

When all other chunks have been parsed, LibPNG starts reading pixel data from IDAT chunks. For each image row, LibPNG allocates and initializes a buffer (lines 31-38 in *png_read_start_row*). This is the faulty function. Specifically, the existence of tRNS chunk and the improper validation of large image width leads to an integer overflow while LibPNG is calculating buffer size for each row (as simplified in *calculateBufSize* at line 35). Because of that the allocated buffer is much smaller than required (line 36). As a consequence, a buffer overflow occurs in *png_memset* causing the program to crash. Notice that the third argument for the function call *memset* (*ptr→rowbytes*) is much larger than the size of the buffer.

2.2 Exposing Vulnerabilities

2.2.1 Traditional Whitebox Fuzzing

Given a benign PNG file having the required data chunks in Figure 1 and the dangerous location in *png_memset*, a Whitebox Fuzzing (TWF) tool can automatically generate an input that exposes the vulnerability. However, suppose the benign file is missing the tRNS chunk, it will be an obstacle for TWF because it is very unlikely that TWF can correctly synthesize the missing chunk and keep the file valid. In fact, if there is no tRNS chunk, the true branch of the IF-statement in line 25 of Listing 1 is not taken. Although TWF can negate the branch and get a chunk with the name "tRNS", its size and content still adheres to specification of another chunk. Where LibPNG expects the size, data, and checksum of the new tRNS chunk, it only finds "random noise". So, TWF overrides perfectly encoded image data only to spend *substantial time* constructing a valid tRNS chunk in its place. Since IDAT chunk is compulsory, TWF spends even more time navigating the space of invalid inputs to construct another IDAT chunk until it finally constructed a valid file that contains a valid tRNS chunk and all compulsory chunks where all integrity constraints are satisfied.

2.2.2 Model-Based Whitebox Fuzzing

We propose Model-based Whitebox Fuzzing (MoWF) as a marriage of model-based backbox fuzzing and whitebox fuzzing. The model-based approach allows MoWF to cover the search space of *valid* test inputs efficiently while the whitebox approach in detail covers each subdomain more effectively. Both approaches are integrated in a feedback loop that is described in Figure 2.

Setup. In this example, the user provides the buggy VLC binary, a crash report, a set of existing benign PNG files (if available) and a PNG model as shown in Listing 2. To implement MoWF, we leverage a *model-based* blackbox fuzzer. The Peach framework allows to specify a file format as Peach Pit [4]. It describes the types of and relationships (size, count, offsets) between data chunks and fields. It also supports fixups and transformers. Fixups allow to repair related data fields, such as checksums. Transformers are used for encoding, decoding and compression.

The PNG Peach Pit in Listing 2 first specifies the generic data chunk (lines 1-14). PNG chunks all contain at least three data fields, specifying the length, type, and checksum of the data chunk. The other data chunks inherit these attributes (lines 15-31), fix the chunk type as enumerable (IHDR, PLTE, tRNS, ..), and add further data fields. The whole PNG file is specified last (lines 32-42). It starts with a specific magic number (Signature for PNG files), followed by a header chunk (IHDR) and upto 30,000 chunks (in flexible order) before ending up with an IEND chunk.

```
1  <DataModel name="Chunk">
2    <Number name="Length" size="32" >
3      <Relation type="size" of="Data" />
4    </Number>
5    <Block name="TypeData">
6      <Blob name="Type" length="4" />
7      <Blob name="Data" />
8    </Block>
9    <Number name="crc" size="32" >
10     <Fixup class="Crc32Fixup">
11       <Param name="ref" value="TypeData"/>
12     </Fixup>
13   </Number>
14 </DataModel>
15 <DataModel name="Chunk_IHDR" ref="Chunk">
16   <Block name="TypeData">
17     <String name="Type" value="IHDR" />
18     <Block name="Data">
19       <Number name="width" size="32" />
20       <Number name="height" size="32" />
21       ...
22     </Block>
23   </Block>
24 </DataModel>
25 ...
26 <DataModel name="Chunk_tRNS" ref="Chunk">
27   <Block name="TypeData">
28     <String name="Type" value="tRNS" />
29     <Blob name="Data" />
30   </Block>
31 </DataModel>
32 <DataModel name="PNG">
33   <Number name="Sig" value="89504e..." />
34   <Block name="IHDR" ref="Chunk_IHDR"/>
35   <Choice name="Chunks" maxOccurs="30000">
36     <Block name="PLTE" ref="Chunk_PLTE"/>
37     ...
38     <Block name="tRNS" ref="Chunk_tRNS"/>
39     <Block name="IDAT" ref="Chunk_IDAT"/>
40   </Choice>
41   <Block name="IEND" ref="Chunk_IEND"/>
42 </DataModel>
```

Listing 2: PNG input model as Peach Pit

Given the setup, to generate the crashing input in the motivating example, MoWF manages to (i) insert a tRNS chunk into proper position in a benign PNG file, (ii) explore the paths affected by the existence of tRNS towards crash location, and (iii) generate specific value for the image width data field in IHDR chunk. This is achieved in four steps.

Step 1. Seed selection and file cracking. As shown in Figure 2, MoWF first selects as *initial input* that file which is closest to a potential crash location. All other PNG files are considered donors, disassembled by the *file cracker* and added to the *fragment pool*. File fragments can be transplanted into input files as needed. If no initial files are provided, MoWF instantiates the initial input from the input model. Then, MoWF marks as *symbolic* all data fields which the user specified as "modifiable". Only modifieable data fields are considered for the fuzzing. In this example, all data fields (e.g., image width) are marked as modifiable except for the chunk's checksum and size. The resulting hybrid symbolic PNG file (i.e., some parts are symbolic where others are concrete) is then executed concolically by a traditional whitebox fuzzer.

Step 2. Adding and removing data chunks. Certain branches in a file-processing program are exercised only if a certain *data chunk* is absent or present. To exercise these branches during path exploration, MoWF removes the specific chunk or adds a new one. First, in the execution of a given file f, MoWF identifies those *crucial if-statements* (IFs) by their dependence on a data field in f of enumerable type. In Listing 1, the IFs in lines 11–26 can be considered crucial while none of the those inside the `handle ****` functions are. In our experiments, we observe that such enumerables do often uniquely identify a data chunk's type. First, MoWF identifies the input bytes in f that influence the outcome of executed branch predicates using classical *taint analysis*. In our example, MoWF determines the relationship between the input bytes above the grey boxes in Figure 1 and the IFs in Listing 1. Then, MoWF learns the type of the referenced data field using the input model. Finally, if the data field is of enumerable type and the IF is not already executed in both directions, then the IF is considered crucial and MoWF removes the corresponding data chunk or adds a new one through transplantation or instantiation from the input model.

Once MoWF identifies the type corresponding to the data chunk being removed or added, the *file stitcher* coordinates the data chunk transplantation. First, the stitcher searches the fragment pool for candidate data chunks that are allowed (according to the input model) to be put at the same level as the chosen chunk in the current seed file f. Finally, the file sticher uses the input model to identify the set of input bytes corresponding to each candidate data chunk in the pool and transplants them into the appropriate location of the receiving file f to generate a number of new seed files, one for each chunk. For our example, in what follows we assume that the candidate containing the tRNS chunk is chosen next.

Step 3. Changing data fields in inserted data chunk. Other branches in a file-processing program are exercised only if specific values are set in the chunks' *data fields*. In our example, the vulnerability is exposed only when the image width is in a range of certain values. To exercise these branches by finding the specific values is the strength of whitebox fuzzing. Selective symbolic execution explores the local search space of semi-valid inputs starting from the negated crucial branch. This local search is very efficient when compared to classical TWF. During exploration, any integrity check is identified and ignored. The potentially invalid files are later fixed during the *file repair*. Once the target location is reached, the whitebox fuzzer checks the satisfiability of the conjunction of path constraint and crash condition (inferred from the given crash report or provided as output of static analysis tool). If the conjunction is satisfiable, the whitebox fuzzer generates a crashing input. Otherwise, it uses the unsatisfiable core to guide the path exploration towards the crash location and does the check again.

Step 4. Repeat. Data chunks can be nested in certain file formats (such as WAV). Thus, MoWF uses the generated files as new seeds to continue the next iteration starting from Step 1. From the augmented seeds (initial seeds + new seeds), MoWF selects a file which is closest to the crash location and moves to next steps. MoWF executes selected file, identifies crucial if-statements, transplants data chunks and continues path explorations.

Summary. In this motivating example, MoWF follows these four steps. During concolic execution, it identifies line 25 (Listing 1) as crucial if-statement. From the input model, the *file stitcher* infers that a tRNS chunk is a candidate for transplantation and it is allowed after PLTE and before the IDAT chunk. So, *file stitcher* transplants a tRNS chunk from the fragment pool or directly instantiates a minimal tRNS chunk from the input model and places it right before IDAT chunk. As a result, the true branch of the if-statement in line 25 is taken and the tRNS chunk is parsed before doing further processing. Once the crash location is reached, the image-width dependent crash condition is checked and a PNG file is produced. The resulting file is still invalid because the new value of image width invalidates the checksum of IHDR chunk. So, the file repair tool fixes the checksum and the vulnerability is exposed.

3. MODEL-BASED WHITEBOX FUZZING

Algorithm 1 gives an overview of the procedure of directed model-based whitebox fuzzing. It takes a program $\mathcal{P}$, an input model $\mathcal{M}$, a set of target locations L in $\mathcal{P}$, and seed inputs T. The *objective* of Algorithm 1 is to generate valid (crashing) files that exercise L. If no target is provided, MoWF uses static analysis to identify dangerous locations in the program, such as locations for potential null pointer dereferences or divisions by zero (line 1-3). The algorithm uses the provided test cases T as seed inputs for the test generation. However, if no seed file is provided, MoWF leverages the input model $\mathcal{M}$ to instantiate a seed file (lines 4-7).

Algorithm 1 Model-Based Whitebox Fuzzing

Input: Program $\mathcal{P}$, Input Model $\mathcal{M}$
Input: Initial Test Suite T, Targets L
Output: Augmented Test Suite T'

```
1: if L = ∅ then
2:     L ← IDENTIFYCRITICALLOCATIONS(P)
3: end if
4: if T = ∅ then
5:     t ← INSTANTIATEASVALIDINPUT(M)
6:     T ← {t}
7: end if
8: while timeout not exceeded do
9:     Target location l ← CHOOSETARGET(L)
10:    Input file t ← CHOOSEBEST(T, l)
11:    Fragment Pool Φ ← FILECRACKER(T, M)
12:    Crucial IFS Λ ← DETECTCRUCIALIFS(t, l, P, M)
13:    for all λ ∈ Λ do
14:        Valid files T_λ ← FILESTITCHER(t, λ, Φ, M)
15:        for all t_λ ∈ T_λ that negate λ do
16:            Hybrid file t̂_λ ← MARKSYMBOLICVARS(t_λ, M)
17:            Files F ← PATHEXPLORATION(t̂_λ, λ, l, L, P)
18:            for all f ∈ F do
19:                Valid file f' ← FILEREPAIR(f, M)
20:                T ← T ∪ f'
21:            end for
22:        end for
23:    end for
24: end while
25: T' ← T
```

The main loop of Algorithm 1 is shown in lines 8-24. First, MoWF chooses the next target location l. If MoWF works in crash reproduction mode, l is the known crash location extracted from the given crash report. Otherwise, l is picked if its average distance to all seed inputs in T is smallest. The distance between an input t and a program location l is specified in Definition 1. Second, MoWF chooses the next seed file t according to a search strategy that seeks to generate the next input with a reduced distance to l (line 10). The remaining seed files are sent to the file cracker to construct the fragment pool Φ in line 11. The fragement pool takes a central role during data chunk transplantation.

Definition 1 (*Input Distance to Location*)

Given an input t, a program $\mathcal{P}$ and a program location l in $\mathcal{P}$. Let $\Omega(t)$ be the set of nodes in the Control Flow Graph (CFG) of $\mathcal{P}$ that are exercised by t. The distance $\delta(t, l)$ from t to l is the number of nodes on the shortest path from any $b \in \Omega(t)$ to l.

Next, Algorithm 1 executes t on $\mathcal{P}$ to determine crucial IFs Λ (line 12). As specified in Definition 2, a crucial IF is evaluated in different directions only depending on the type of the data chunks present in t. Our implementation leverages $\mathcal{M}$ to identify crucial IFs by their dependence on a data field in t of enumerable type. We observed that such enumerables do often uniquely identify a data chunk's type. Note that we ignore executed IFs negating which does not reduce the distance to the target location l.

Definition 2 (*Crucial IF-statement*)

Given input t for program $\mathcal{P}$ and a target location l in $\mathcal{P}$, an if-statement b in $\mathcal{P}$ is crucial *if*
1) the statement b is executed by t in $\mathcal{P}$,
2) only one direction of b has been taken,
3) the negation of the branch condition at b reduces the distance to l, and
4) let $\varphi(b)$ be the branch condition at b; the outcome of $\varphi(b)$ depends on a field in t that specifies the chunk's type.

For each crucial IF λ thus identified, Algorithm 1 employs the file stitcher to negate λ's branch condition (lines 13-14). For each stitched file t_λ that successfully negates λ, the algorithm executes selective symbolic execution followed by file repair to fine-tune the specific values of the data chunks and reduce the distance to l (lines 15-20). More specifically, it marks all modifiable data fields in t_λ as symbolic and starts the directed path exploration (lines 16-17). During path exploration, MoWF does not collect integrity checks as branch constraints. For instance, a checksum check might not allow to change a data field which would otherwise lead to reducing the distance to L (cf. TaintScope [26]). Such integrity constraints are repaired in line 19. Whenever a potential dangerous location in L is reached, MoWF checks if the crash condition is satisfied and generates a crashing test case accordingly.

A detailed discussion of the procedures in Algorithm 1 is found in the following sections:

#	Procedure	Discussion
2	IDENTIFYCRITICALLOCATIONS	§3.1
5	INSTANTIATEASVALIDINPUT	§3.2
10	CHOOSEBEST	§3.1
11	FILECRACKER	§3.2
12	DETECTCRUCIALIFS	§3.2
14	FILESTITCHER	§3.2
16	MARKSYMBOLICVARS	§3.3
17	PATHEXPLORATION	§3.3 & §3.4
19	FILEREPAIR	§3.2

3.1 Directed Model-Based Search

In order to generate inputs that expose vulnerabilities, MoWF uses the initial seed inputs T to reduce the distance to the provided or identified critical location l until it is reached and the crash condition is satisfied.

Critical Locations. If no targets L are provided to the algorithm, MoWF identifies critical locations in the program $\mathcal{P}$. A *critical location* is a program location that may expose a vulnerability if exercised by an appropriate input. There are several methods to identify such critical locations [13, 26]. In our implementation, we use IDAPro [2] to dissamble the program binary $\mathcal{P}$ and perform some lightweight analysis to identify instructions that conform to the patterns shown in Listing 3. These patterns partially cover program instructions that may trigger divide-by-zero and null-pointer dereference vulnerabilities. Specifically, we focus on division and memory move instructions taking registers or stack arguments as operands. For those instructions, the crash condition is obvious. Once a critical location is reached during concolic exploration, we just check whether the value of register/stack argument is zero (in case it is concrete) or can be zero (in case it is symbolic).

```
div   register
div   [ebp + argument_offset]
mov   operand, [register]
mov   operand, [ebp + argument_offset]
mov   [register], operand
mov   [ebp + argument_offset], operand
```

Listing 3: Crash instruction templates

Model-based Search. To generate input that reduces the distance to l, MoWF first chooses the seed input t with the least distance to l and then identifies the executed crucial IFs Λ (lines 10, 12 in Alg. 1). The task of the subsequent data chunk transplantation and instantiation will be to generate valid inputs that negate the branch conditions of Λ. While other implementations are possible, we decided to implement a hill climbing algorithm. Our implementation of CHOOSEBEST selects the input file $t \in T$ such that for selected location $l \in L$ we have that the distance from t to l is minimal. To detect crucial branches Λ, MoWF first determines, using taint analysis, those input bytes in t that may impact the outcome of some $b \in \Omega(t)$. We recall that $\Omega(t)$ is the set of nodes in the CFG of program $\mathcal{P}$ which are exercised by t. In our implementation of DETECTCRUCIALIFS, we leverage those capabilities in a symbolic execution tool, Hercules. Next, MoWF uses the CFG to compute the number of nodes on the shortest path between b and location $l \in L$. The negation of $\varphi(b)$ may reduce the distance to l only if b is in static backward slice of l and the branch b' immediately following b does not have a smaller number of nodes on the shortest path between b' and l. Lastly, MoWF uses $\mathcal{M}$ to determine the data field corresponding to the identified input bytes and whether the data field specifies the chunk's type. If all conditions specified in Definition 2 are met, then b is marked as a crucial IF and added to Λ.

3.2 Transplantation, Instantiation, and Repair

File Cracker. "File cracking" refers to the process of interpreting valid files according to a provided input model (i.e., the Peach Pit file). Given the input model $\mathcal{M}$ and a valid file $t \in T$, the FILECRACKER identifies all data chunks and their data fields in t. In model-based blackbox fuzzers like Peach Fuzzer [3], the valid input files are cracked and fuzzed independently. However, in MoWF we crack all files and place their data components inside a *fragment pool*. As a result, we can consider all files (and even the input model) as donors for data transplantation. By doing that, MoWF can generate more (semi) valid files and improve coverage.

File Stitcher. Given a valid file t and the crucial IF λ, the objective of FILESTITCHER is to negate $\varphi(\lambda)$ and reduce the distance to l by adding or removing chunks from t. First, the stitcher has to determine the chunk c in t that should be removed or before which a different chunk should be added in order to negate $\varphi(\lambda)$. Chunk c was memorized previously when determining that the outcome of λ depends on the data field specifying c's type. Second, the stitcher generates a new file by removing c from t if allowed according to $\mathcal{M}$. Third, for each chunk type $\mathcal{C}$ that is allowed before c in t:
i) *Transplantation.* If there exists a chunk c' of type $\mathcal{C}$ in the pool Φ, copy the input bytes corresponding to c' from the donor file to the position before c in the receiving file t.
ii) *Instantiation*. Otherwise, use the specification of $\mathcal{C}$ in $\mathcal{M}$ as a template to generate the bytes for c' before c in t. All files thus generated that actually negate λ will be used for the subsequent selective symbolic execution stage.

File Repair. Given a file f and the input model $\mathcal{M}$, the file repair tool re-establishes the integrity of the file. Our implementation utilizes the fixup and transformers that can be specified in $\mathcal{M}$ in the Peach framework.

3.3 Selective and Targeted Symbolic Execution

We reuse the targeted search strategy for symbolic exploration implemented in Hercules [22]. Basically, to mitigate the path explosion problem, it enables fully symbolic reasoning only in some selected modules of interest (i.e., executable binaries like .exe and .dll files). The list of selected modules can be inferred from the target module TM, which contains the selected target location, and a so-called Module Dependency Graph (MDG). The MDG is constructed by running the program under test with benign inputs and collecting the control transfer between program modules. Using the constructed MDG, TM and all modules on paths from entry module (main program) to TM are selected to explore in fully symbolic execution mode.

The search strategy of Hercules is targeted in the sense that it explores program paths towards a target location (critical locations like crashing one) by pruning irrelevant paths. Moreover, Hercules leverages the unsatisfiable core produced by a theory prover like Z3 [12] to guide the exploration.

3.4 Handling Incomplete Memory Modeling

The memory models of symbolic execution engines, like Hercules, KLEE or S2E [22, 10, 11], do not support memory allocation with symbolic size. If a symbolic size is given, it is concretized before allocating heap memory. The concretization mechanism could prevent us from exposing heap buffer overflow vulnerabilities. Suppose in the motivating example the image width of the benign PNG file is very small, say 1, and it is marked as symbolic. In the processing code, LibPNG needs to allocate a heap buffer having symbolic size that depends on *width* (and other symbolic variables). When the buffer is allocated, *width* is bound in PC by the constraint on concretized value for allocated buffer size.

Once the crash location (e.g., the instruction accessing the allocated heap buffer) is reached, Hercules checks the satisfiablity of the conjunction between the current path constraint PC and the crash condition CC. Suppose that to satisfy the crash condition, the image width must be large enough. For the current file with the small image width, the crash condition CC could contradict the path constraint PC; $PC \wedge CC$ is unsatisfiable. Usually, based on the unsatisfiable core[1] of $PC \wedge CC$, Hercules find a set of branches that can be negated to explore neighboring paths along which the crash condition CC may be satisfiable. However, since *width* is already bound, there exists no alternative path along which the crash condition CC can be satisfied.

In our extension of Hercules, we leverage recent advances in maximal satisfication with Z3 (MaxSMT)[8, 12]. Using a whitelist, our tool automatically marks certain clauses as "soft clauses". The Satisfiability Modulo Theory (SMT) Solver Z3 allows to generate an assignment to the symbolic variables as solution that satisfies the conjunction of all clauses – but not necessarily the soft clauses.

Specifically, in our case we set all constraints in CC as hard clauses while specifying e.g., constraints due to memory allocation in PC as soft clauses. To identify which constraints in PC can be soft, first we check whether the conjunction $PC \wedge CC$ is unsatisfiable. If so, we extract all symbolic variables in CC. Thereafter, we iterate through all constraints in PC and consider them as soft constraints accordingly if they contain any symbolic variable from CC. After all these steps, we get PC', the updated PC, and we send another query to MaxSMT solver to check the maximum satisfiability of $PC' \wedge CC$. If $PC' \wedge CC$ is satisfiable (by possibly

[1]Given an unsatisfiable Boolean propositional formula in conjunctive normal form, a minimal subset of clauses whose conjunction is still unsatisfiable is called an unsatisfiable core of the original formula.

making one or more soft clauses in PC' as false) – we generate a input file as the solution to the constraint. As an additional confirmation, we validate the generated file by feeding it to the program binary and checking whether it crashes the program.

4. IMPLEMENTATION

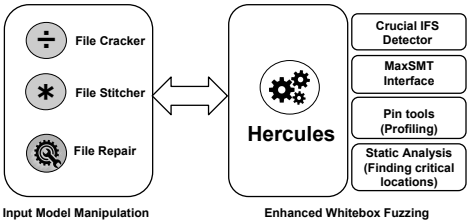

Figure 3: Components of our MoWF tool

Our MoWF tool is based on several third-party tools and libraries. We implemented our technique into the `Hercules` [22] directed symbolic execution engine which itself leverages `S2E` [11] and the `Z3` [12] satisfiability modulo theory constraint solver. We also improved the accuracy of the taint analysis that is implemented in `Hercules`. `IDAPro` [2] and the Intel Dynamic Binary Instrumentation Tool [20] (or `PIN tool`) were used for static analysis to find dangerous locations in the program code executing which might crash the program (cf. §3.1). The `PIN tools` were also used i) for instruction profiling to generate the execution trace and compute the distance of the current seed input to the dangerous locations, and ii) for branch profiling to determine which crucial branches are explored. The framework around the `Peach` model-based blackbox fuzzer [3] allowed us to implement the input model-based components such as *File cracker*, *File Stitcher* and *File Repair*. In fact, the first was modified for our purposes and the latter two were implemented from scratch, for instance, to support data chunk transplantation.

5. EXPERIMENTAL EVALUATION

We evaluated our MoWF technique experimentally to answer the following research questions.

- **RQ.1** How many vulnerabilities are exposed by MoWF compared to Traditional Whitebox Fuzzing (TWF)?
- **RQ.2** How many vulnerabilities are exposed by MoWF compared to Model-based Blackbox Fuzzing (MoBF)?
- **RQ.3** How many vulnerabilities are exposed by MoWF if no initial seed inputs are available?

Each technique was evaluated with a 24 hour time budget.

5.1 Experimental Setup

5.1.1 Subjects

We selected our subjects from a pool of well-known program binaries of video players, document readers, music players, and image editors – which take a variety of complex file formats. Since Hercules serves as a base line technique, we also added all five subjects on which Hercules was evaluated originally [22] (shown with grey background). We also took the categories of vulnerabilities into consideration. As shown in Table 1, we chose eight distinct real-world applications (some with different versions): Adobe Reader (AR)[2], Video Lan Client (VLC)[3], Windows Media Player (WMP), Real Player (RP)[4] and Music Animation Machine MIDI Player (MP)[5], XnView (XNV)[6], LibPNG (LTP)[7] and Orbital Viewer (OV)[8].

Table 1: Subject Programs

Program	Version	Buggy module	Size	Errors
Video Lan Client	2.0.7	libpng.dll	184 KB	1
Video Lan Client	2.0.3	libpng.dll	182 KB	1
Libpng Test Program	1.5.4	libpng.dll	176 KB	1
XnView	1.98	XnView.exe	4.46 MB	0 + 3
Adobe Reader	9.2	cooltype.dll	2.32 Ml	1
Windows Media Playe	9.0	quartz.dll	1.22 Ml	2 + 1
Real Player SP	1.0	realplay.exe	60 KB	1
MIDI Player	0.35	mamplayer.ex	336 KB	1
Orbital Viewer	1.04	ov.exe	538 KB	1
			Total:	9 + 4

Table 1 shows not only the subjects and their versions but also the target buggy modules and their respective sizes. In addition, it features the number of known vulnerabilities that we sought to reproduce. In one case (XnView), we started without any known vulnerabilities and looked for unknown ones. In other cases, although we targeted the known vulnerabilities, we managed to discover new ones. Indeed, our MoWF tool reproduced successfully all 9 known errors and discovered 4 unknown errors – 3 in XnView and 1 in Windows Media Player (See Section 5.2).

5.1.2 Input Modeling

To *define input models* of five file formats (PDF, PNG, MIDI, FLV and ORB) from scratch, we utilized the modeling language of the Peach model-based blackbox fuzzer. We augmented the input model for WAV files which is provided freely by Peach Fuzzer. In particular, we modeled one common image file (PNG), three audio and video files (MIDI, WAV and FLV), one portable document file (PDF) and one geometry file (ORB). In Table 2, we report the size of the input models which are relatively small – ranging from 4 KB to 14 KB. It took us less than a day to write each model for a file format.

Table 2: Information on the Input Models

Format	Size	Time spent	#Files	Average size
PDF	4.5 KB	12 hours	10	200 KB
PNG	8.3 KB	4 hours	10	55 KB
MIDI	13.9 KB	4 hours	10	20 KB
FLV	6.0 KB	4 hours	10	300 KB
ORB	6.0 KB	8 hours	10	4 KB
WAV*	7.5 KB	2 hours	10	260 KB

5.1.3 Initial Seed Files Selection

To *select the initial seed files*, we randomly downloaded 10 files of the corresponding format from the Internet, except ORB and PNG initial seed files. The ORB files were downloaded from software vendor's website[9] while PNG files were downloaded from the Schaik online test suite.[10] The average size of seed files in each test suite is shown in the fifth column of Table 2.

[2]https://get.adobe.com/reader/
[3]http://www.videolan.org/index.html
[4]http://www.real.com/sg
[5]http://www.musanim.com/player/
[6]http://www.xnview.com/en/
[7]http://www.libpng.org/pub/png/libpng.html
[8]http://www.orbitals.com/orb/ov.htm
[9]http://www.orbitals.com/orb/ov.htm
[10]http://www.schaik.com/pngsuite

5.1.4 Infrastructure

We evaluated three tools, our MoWF tool, the Hercules Traditional Whitebox Fuzzer (TWF) and the Peach Model-Based Blackbox Fuzzer (MoBF). For the experiments, we used the community version of Peach Fuzzer which is provided with its source code.[11] Both model-based techniques used the same input models. All subject programs were run on Windows XP 32-bit SP 3. For each program, each tool was configured for a timeout after 24 hours of execution. We conducted all experiments on a computer with a 3.6 GHz Intel Core i7-4790 CPU and 16 GB of RAM.

5.2 Results and Analysis

Table 3: The vulnerabilities exposed by our MoWF tool, the Hercules TWF, and the Peach MoBF. Vulnerabilities from the Hercules benchmark are marked as grey.

Program	Advisory ID	Model	Files	MoWF	MoBF	TWF
VLC 2.0.7	OSVDB-95632	PNG	10	✔	✘	✘
VLC 2.0.3	CVE-2012-5470	PNG	10	✔	✘	✘
LTP 1.5.4	CVE-2011-3328	PNG	10	✔	✘	✘
XNV 1.98	Unknown-1	PNG	10	✔	✔	✘
XNV 1.98	Unknown-2	PNG	10	✔	✔	✘
XNV 1.98	Unknown-3	PNG	10	✔	✔	✘
WMP 9.0	Unknown-4	WAV	10	✔	✔	✘
WMP 9.0	CVE-2014-267	WAV	10	✔	✘	✔
WMP 9.0	CVE-2010-071	MIDI	10	✔	✘	✔
AR 9.2	CVE-2010-220	PDF	10	✔	✘	✔
RP 1.0	CVE-2010-300	FLV	10	✔	✘	✔
MP 0.35	CVE-2011-050	MIDI	10	✔	✔	✔
OV 1.04	CVE-2010-068	ORB	10	✔	✔	✔

Table 3 shows the results in reproducing known vulnerabilities and finding unknown ones of the three compared techniques. Overall, in the experiments our MoWF tool outperforms both Hercules and Peach. While our MoWF tool successfully generated 13 crash-inducing inputs, neither Hercules nor Peach can produce half of them . Furthermore, our MoWF tool also found potential unknown vulnerabilities in Windows Media Player and XnView. Indeed, these vulnerabilities have previously not been reported at MITRE[12], OSVDB[13] or Exploit-DB.[14]. In addition, the power of our MoWF tool is also demonstrated by its ability to expose different types of vulnerabilities including integer and buffer overflows, null pointer dereference and divide-by-zero. In the following sections, we have an in-depth analysis to answer the three research questions about the effectiveness and sensitivity of our approach.

RQ.1 Versus Traditional Whitebox Fuzzing

Our experiments confirm the observations that TWF is unlikely to synthesize missing composite data chunks. As in OSVDB-95632, CVE-2012-5470, CVE-2011-3328 and Unknown 1-4, Hercules cannot produce crash inputs to expose the vulnerabilities because they require the existence of optional composite data chunks. In our experiments, Hercules gets stuck in synthesizing such required data chunks. In particular, the following requirments must be met to expose the 7 vulnerabilities that are not in the Hercules benchmark:

OSVDB-95632 (Buffer Overflow): It requires a PNG file with a tRNS optional data chunk specifing either alpha values that are associated with palette entries (for indexed-colour images) or a single transparent colour (for greyscale and truecolour images). Moreover, the value of a data field (image width) in IHDR chunk (the header chunk of PNG) must be able to trigger an integer overflow in the LibPNG plugin in VLC 2.0.7.

CVE-2012-5470 (Buffer Overflow): It requires a PNG file with a tEXt optional data chunk which stores text strings associated with the image, such as an image description or copyright notice. Furthermore, the length of the data chunk must be big enough to exceed the size of a heap buffer allocated for the image. However, it cannot be so huge that it prevents LibPNG from successfully allocating a heap buffer that is supposed to store the data in tEXt chunk.

CVE-2011-3328 (Divide-by-Zero): They require a PNG file with a cHRM optional data chunk. The cHRM specifies chromaticities of the red, green, and blue display primaries used in the image, and the referenced white point. Second, some data fields in cHRM chunk must have specific values to trigger a divide-by-zero bug in the LibPNG library.

Unknown 1-3 (Memory Read Access Violation): They require PNG files having optional data chunks (iTXt, zTXt or iCCP accordingly) which have no content. That is, the chunks that specify a size of zero followed by chunk name and checksum.

Unknown 4 (Divide-by-Zero): It requires a WAV file in which the format chunk contains an optional extra composite data field and one specific byte in the field is zero.

Unlike Hercules, our MoWF tool leverages the input models to transplant required data chunks from other files in the initial test suite or generate the chunks automatically from the input model. Hence, our MoWF tool can successfully produce crash inputs as witnesses for the seven vulnerabilites mentioned above.

Since our MoWF tool is an extension of Hercules, it can successfully reproduce all six vulnerabilites in the Hercules benchmark. As we will see for RQ.3, our MoWF tool does not require seed inputs to reproduce three out of the six vulnerabilites in the Hercules benchmark (CVE-2010-0718, CVE-2011-0502 and CVE-2010-0688) because of its capability to generate (semi-) valid files directly from input models.

RQ2. Versus Model-Based Blackbox Fuzzing

The Peach model-based blackbox fuzzer cannot expose half of the vulnerabilities that our MoWF tool can expose (see Table 3). We note that we conservatively assume that data chunk transplanation and instantiation is available in Peach – even though it is not. It is worth mentioning that supporting transplanation and instantiation in Peach could be challenging. In fact, finding the correct chunk to transplant and transplanting it to the correct location in the seed input is subject to combinatorial explosion in an undirected fuzzing technique like Peach. In constrast, MoWF uses information about crucial IFs to direct the transplantation.

In the experiments, we simulated Peach's capability to do data chunk transplanation and instantiation by augmenting the set of all 10 seed inputs where none contains the missing data chunk with at least one seed input where we manually transplanted the missing data chunk. In Table 3, we indicate that Peach (with the simulated capability) can expose three vulnerabilities Unknown 1-3 since these only require the existence of empty-data optional chunks.

However, for the remaining 10 vulnerabilities, the MoBF tool Peach cannot successfully expose 7 of 10 vulnerabilities even though we provide inputs with the required optional data chunks. It is because of its *limitation on generating specific values*. The reason lies with the inability of blackbox fuzzing to generate the specific values for data fields that would expose deep vulnerabili-

[11] http://community.peachfuzzer.com to download.
[12] http://cve.mitre.org/
[13] http://osvdb.org/
[14] https://www.exploit-db.com/

ties. For example, given a 4-byte integer data field, the chance for a blackbox fuzzer to randomly mutate and get a specific value X is extremely small, just only $1/2^{32}$. In contrast, symbolic execution-based whitebox fuzzing is very good at finding such values.

Meanwhile, our MoWF tool is an enhancement of TWF (by leveraging input models) and can tackle both the missing data chunk problem and the limitation on generating specific input values. As a result, it can successfully produce test cases to expose all of the 13 vulnerabilities.

RQ3. Sensitivity to the Initial Test Suite

Table 4: Vulnerabilities exposed by our MoWF tool if no initial seed files are provided.

Program	Advisory ID	Model	#Files	MoWF
VLC 2.0.7	OSVDB-95632	PNG	0	✔
VLC 2.0.3	CVE-2012-5470	PNG	0	✔
LTP 1.5.4	CVE-2011-3328	PNG	0	✔
XNV 1.98	Unknown-1	PNG	0	✔
XNV 1.98	Unknown-2	PNG	0	✔
XNV 1.98	Unknown-3	PNG	0	✔
WMP 9.0	Unknown-4	WAV	0	✘
WMP 9.0	CVE-2014-2671	WAV	0	✘
WMP 9.0	CVE-2010-0718	MIDI	0	✔
AR 9.2	CVE-2010-2204	PDF	0	✘
RP 1.0	CVE-2010-3000	FLV	0	✘
MP 0.35	CVE-2011-0502	MIDI	0	✔
OV 1.04	CVE-2010-0688	ORB	0	✔

For this experiment, we run our MoWF tool with no initial seed inputs as shown in Table 4. By leveraging input models of PNG, MIDI and ORB, for each file format our MoWF automatically generates one minimal seed file. In particular, a minimal PNG file is an 1x1 image having four mandatory chunks – IHDR, PLTE, IDAT and IEND. In case of MIDI, it is a single track audio file with one header chunk (MThd) and one audio track chunk (MTrk). The minimal ORB file contains all required properties for rendering an orbital object. Once the files are generated, we run our MoWF tool on all subjects listed in Table 4.

The experiments show that with the minimal files, our MoWF tool can expose 9 of 13 vulnerabilities (which can be revealed by PNG, MIDI and ORB files) as reported in Table 3. It means that our MoWF tool exposes 70% vulnerabilites without any provided seed inputs providing evidence that MoWF technique reduces the dependence of TWF on selected seed inputs.

MoWF does not succeed in exposing the vulnerabilities in 4 of 13 vulnerabilities because they require WAV, FLV and PDF files as inputs. However, our models for these file formats are still coarse. Although they are enough to allow MoWF to work with given test suites, they need to be more complete to support directly generating (semi-) valid files. Since these file formats are complex, on one hand we can spend more time to read and fully understand their specifications in order to augment the input models. On the other hand, we can reuse exhaustive models written by software vendors or the owners of file formats. For instance, according to a post at the official Adobe Blog,[15] developers at Adobe System wrote their model for PDF file (which was a proprietary format controlled by Adobe until 2008) and used Peach Fuzzer to fuzz their most popular software – Adobe Reader. Given such (partially) complete input models, our MoWF approach would complement MoBF tool like Peach Fuzzer to maximize the utility of these models and hence expose more vulnerabilities.

[15]https://blogs.adobe.com/security/tag/fuzzing

6. THREATS TO VALIDITY

The main threat to external validity is the generality of our results. MoWF has been developed for real-world program binaries that take complex program inputs. We choose a variety of well-known programs from different domains where specifications of the input models are available. While for proprietary applications such format specifications might not be available, we believe that grammar inference techniques can be a powerful tool to automatically derive the input model. Half of the vulnerabilities have already been picked in earlier work [22]. To showcase the effectiveness of MoWF, the other half has been chosen such that an optional data chunk is required to expose the vulnerability.[16]

The main threat to internal validity is selection bias during the seed selection (see Table 2). We chose the seed inputs either randomly from a benchmark or from the internet. Moreover, our experiments confirm the reduced dependence on the available seed inputs.

The main threat to construct validity is the correctness of our implementation. However, our tool is an extension of both Hercules and Peach, the two baselines for our evaluation. So, our tool inherits the incorrectness of the baseline.

7. RELATED WORK

The first automated testing technique for file-processing programs FUZZ was implemented in 1990 by Miller et al. [21] to understand the reliability of UNIX tools. Since then fuzzing has evolved substantially, become widely adopted into practice, and exposed serious vulnerabilities in many important software programs. A *fuzzer* quickly generates an excessive amount of program inputs in an attempt to make it crash. Today, most compilers support to inject so-called sanitizers during the compilation of the program under test. A *sanitizer* is an automated oracle that can expose more intricate but serious software bugs, such as buffer overflows, data races, and memory errors. Together, fuzzers and the sanitizers allow the automated testing and exposing of deep and intricate software bugs in programs of any scale. Security sensitive programs are *hardened* in a feedback loop where a program is first sanitized, then fuzzed, the exposed errors fixed, the patched version is fuzzed again, and so on.

We can distinguish the more efficient blackbox techniques that generate test inputs without the analysis of the program source code and the more effective whitebox techniques that leverage program analysis to expose bugs hiding deeper in the source code of the program.

Blackbox Fuzzing [3, 5, 6, 7, 21]. Programs processing simply structured plain text input can be fuzzed by random input generators, like FUZZ [21]. In fact, random test generation can be a very efficient test generation technique [9]. However, for programs processing highly structured input files, like a PDF Reader, most random files are rejected as invalid. Hence, model-based blackbox fuzzing (MoBF) tools utilize a user-given input model to generate *valid* random files [3, 5, 6, 7]. However, due to the random choice of values for data fields, MoBF may still be ineffective in exposing more deeper errors in the program's functionality. Systematic path exploration to enumerate the specific values of a data field is significantly more effective.

Traditional Whitebox Fuzzing [15, 22, 13, 26] seeks to explore alternative paths in the program by substituting input bytes in a given file with other values. *Taint-based whitebox fuzzing* [13, 26] identifies those "hot bytes" in the input file that impact the value of a dangerous location, like a divisor or system call. Fuzzing the

[16]See RQ.1. in Section 5.2.

hot bytes can reveal errors more quickly. More effective *symbolic execution-based whitebox fuzzers* substitute input bytes that impact the outcome of a branch with symbolic variables and employ symbolic execution [15, 22] to negate those branches. *Checksum-aware whitebox fuzzing* [26] attempts to identify checksum checks and circumvent them during whitebox fuzzing. The check is identified as the first `if`-statement s that depends on many input bytes and is circumvented by removing s from the program. To repair the generated malformed files, the branch-condition of s is computed in terms of the identified input bytes made symbolic. However, checksum-aware whitebox fuzzing cannot solve any other integrity constraints, like chunk size or offset. Throughout the paper, we have shown the limitations of traditional whitebox fuzzing, such as being bogged down by the large search space of invalid inputs and the dependence in seed files.

Grammar-based Whitebox Fuzzing (GWF) [14] generates inputs that are valid w.r.t. a context-free grammar $\mathcal{G}$. We use the example in Listing 4 for illustration.

```
1  int i;
2  char* input;
3  char getNextToken() {
4    return input[i++];
5  }
6  bool isSorted() {
7    int prev_digit = 0;
8    if ('{' == getNextToken()) {
9      do {
10       char token = getNextToken();
11       if (',' == token) continue;
12       if ('}' == token) return true;
13       int digit = asInt(token);
14       if (prev_digit > digit) return false;
15       prev_digit = digit;
16     } while (true);
17   }
18   return false;
19 }
```

Listing 4: isSorted() returns true if the input is a sorted list of single digit numbers

The context-free grammar $\mathcal{G}$ may be written as

$$\mathcal{G} \rightarrow \{Numbers\} \quad (1)$$
$$Numbers \rightarrow Numbers, Numbers \quad (2)$$
$$Numbers \rightarrow Digit \quad (3)$$
$$Digit \rightarrow \texttt{0} \mid \texttt{1} \mid \texttt{2} \mid \texttt{3} \mid \texttt{4} \mid \texttt{5} \mid \texttt{6} \mid \texttt{7} \mid \texttt{8} \mid \texttt{9} \quad (4)$$

which encodes that valid inputs start with an open curly bracket followed by a comma-separated list of (at least one) digits and a closing curly bracket. GWF encodes a path condition as regular expression. For input `{1,2}`, GWF yields the following constraint R to explore the alternative branch where the input does not end in a curly bracket:

$$\text{token}_1 = \{ \quad (5)$$
$$\wedge\ \text{token}_2 = Digit \quad (6)$$
$$\wedge\ \text{token}_3 = , \quad (7)$$
$$\wedge\ \text{token}_4 = Digit \quad (8)$$
$$\wedge\ \text{token}_5 \neq \} \quad (9)$$

where *Digit* is a symbolic variable. Using a context-free constraint solver, it is possible to derive an array with three digits that is accepted by both G and R (e.g., `{0,0,0}`). However, since the regular expression cannot express the arithmetic relationship between token_2 and token_4 (i.e., $1 < 2$), a completely different path might be exercised. This renders GWF both unsound as well as incomplete. In contrast, MoWF maintains path conditions as SMT formulas and merely *prunes* paths that are exercised by inputs that are invalid w.r.t. the input model. Moreover, the context-free language which encodes the file format cannot express integrity constraints such as the checksum or the size of a data chunk. Functions computed over the data in a data field, such as a compression algorithm, cannot be expressed either. Our input models allow to specify integrity constraints and compression algorithms through the concept of Fixups and Transformers.

Hybrid Fuzzing. Driller [23] combines the effectiveness of whitebox and the efficiency of blackbox fuzzing. After running the blackbox fuzzer for some time, the whitebox fuzzer is run on the most promising seed files produced by the blackbox fuzzer. In contrast to MoWF, Driller does not leverage information from an input model to generate more valid files. Driller does not primarily target such programs that process highly structured inputs. In this respect, our approache is orthogonal to Driller. Driller can benefit from MoWF when testing programs processing highly structured inputs.

8. DISCUSSION

We introduced Model-based Whitebox Fuzzing (MoWF) as an automated testing technique for program binaries that process highly structured inputs. We have observed that certain branches in a file-processing program are exercised only depending on i) the *presence* of a specific data chunk, ii) a *specific value* of a data field in a data chunk, or iii) the *integrity* of the data chunks. Hence, we extend HERCULES an existing traditional whitebox fuzzing technique not only to set specific values of the fields but also to add/remove complete chunks and re-establish their integrity during fuzzing.

Pruning Invalid Paths. We discussed several approaches to prune the vast search space of invalid inputs, including integrity enforcement and data chunk transplantation and instantiation. As opposed to Traditional Whitebox Fuzzing (TWF), MoWF is capable of negating those branches that are exercised only in the presence of certain data chunks without having to iteratively construct the data chunk by exploring the parser code. All generated test inputs are valid in that they adhere to the input model. Integrity constraints are enforced. Given a 24hour time budget, our MoWF tool exposed all of thirteen vulnerabilities in our subject programs while the TWF tool exposed only six.

Reduced Seed Dependence. We also investigated the dependence of MoWF on the provided initial seed files. MoWF can instantiate the initial seed files directly from the provided input model. Moreover, given a seed input that is missing a data chunk to reach a target location, MoWF allows to utilize other seed files as donors, transplant the missing data chunk, and construct a new seed input that is closer to the target location. In the absence of a donor, the missing data chunk can be directly instantiated from the input model. Out of the thirteen vulnerabilities in our experimental subjects our MoWF tool exposed nine *without any seed inputs*.

In summary, MoWF is a promising fuzzing technique for program binaries that process highly structured input. It is particularly helpful when no initial seed files are available that contain the required optional data chunks. Given the same time budget, MoWF can generate more valid test inputs which aids in exposing vulnerabilities that could not be exposed otherwise.

9. ACKNOWLEDGMENTS

We thank Law Wen Yong for his help with implementation. This research was partially supported by a grant from DSO National Laboratories, Singapore, and the National Research Foundation, Prime Minister's Office, Singapore under its National Cybersecurity R&D Program (TSUNAMi project, Award No. NRF2014NCR-NCR001-21) and administered by the National Cybersecurity R&D Directorate.

10. REFERENCES

[1] Specification of the DEFLATE Compression Algorithm. https://tools.ietf.org/html/rfc1951. Accessed: 2016-02-13.

[2] Tool: IDA multi-processor disassembler and debugger. https://www.hex-rays.com/products/ida/. Accessed: 2016-04-04.

[3] Tool: Peach Fuzzer Platform. http://www.peachfuzzer.com/products/peach-platform/. Accessed: 2016-01-23.

[4] Tool: Peach Fuzzer Platform (Input Model). http://community.peachfuzzer.com/v3/DataModeling.html. Accessed: 2016-01-23.

[5] Tool: SPIKE Fuzzer Platform. http://www.immunitysec.com. Accessed: 2016-01-23.

[6] Tool: Suley Fuzzer. https://github.com/OpenRCE/sulley. Accessed: 2016-01-23.

[7] G. Banks, M. Cova, V. Felmetsger, K. Almeroth, R. Kemmerer, and G. Vigna. Snooze: Toward a stateful network protocol fuzzer. In *Proceedings of the 9th International Conference on Information Security*, ISC'06, pages 343–358, 2006.

[8] N. Bjorner and A.-D. Phan. vz - maximal satisfaction with z3. In T. Kutsia and A. Voronkov, editors, *SCSS 2014. 6th International Symposium on Symbolic Computation in Software Science*, volume 30 of *EPiC Series in Computing*, pages 1–9, 2014.

[9] M. Böhme and S. Paul. On the efficiency of automated testing. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 632–642, 2014.

[10] C. Cadar, D. Dunbar, and D. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 209–224, 2008.

[11] V. Chipounov, V. Kuznetsov, and G. Candea. S2e: A platform for in-vivo multi-path analysis of software systems. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 265–278, 2011.

[12] L. De Moura and N. Bjørner. Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340, 2008.

[13] V. Ganesh, T. Leek, and M. Rinard. Taint-based directed whitebox fuzzing. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 474–484, 2009.

[14] P. Godefroid, A. Kiezun, and M. Y. Levin. Grammar-based whitebox fuzzing. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 206–215, 2008.

[15] P. Godefroid, M. Y. Levin, and D. A. Molnar. Automated whitebox fuzz testing. In *Proceedings of the 2008 Network and Distributed System Security Symposium*, volume 8 of *NDSS '08*, pages 151–166, 2008.

[16] I. Haller, A. Slowinska, M. Neugschwandtner, and H. Bos. Dowsing for overflows: A guided fuzzer to find buffer boundary violations. In *Proceedings of the 22Nd USENIX Conference on Security*, SEC'13, pages 49–64, 2013.

[17] F. M. Kifetew, R. Tiella, and P. Tonella. Generating valid grammar-based test inputs by means of genetic programming and annotated grammars. *Empirical Software Engineering*, pages 1–34, 2016.

[18] S. Y. Kim, S. Cha, and D.-H. Bae. Automatic and lightweight grammar generation for fuzz testing. *Comput. Secur.*, 36:1–11, July 2013.

[19] Z. Lin and X. Zhang. Deriving input syntactic structure from execution. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, SIGSOFT '08/FSE-16, pages 83–93, 2008.

[20] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 190–200, 2005.

[21] B. P. Miller, L. Fredriksen, and B. So. An empirical study of the reliability of unix utilities. *Commun. ACM*, 33(12):32–44, Dec. 1990.

[22] V.-T. Pham, W. B. Ng, K. Rubinov, and A. Roychoudhury. Hercules: Reproducing crashes in real-world application binaries. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 891–901, 2015.

[23] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *NDSS '16*, pages 1–16, 2016.

[24] Tool. LibPNG Library. http://www.libpng.org/pub/png/libpng.html. Accessed: 2016-02-13.

[25] Tool. Video Lan Client (VLC). http://www.videolan.org/index.html. Accessed: 2016-02-13.

[26] T. Wang, T. Wei, G. Gu, and W. Zou. Taintscope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, pages 497–512, 2010.

[27] X. Wang, L. Zhang, and P. Tanofsky. Experience report: How is dynamic symbolic execution different from manual testing? a study on klee. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ISSTA 2015, pages 199–210, 2015.