Transformation-aware Exploit Generation using a HI-CFG

Dan Caselden
Alex Bazhanyuk
Mathias Payer
Laszlo Szekeres
Stephen McCamant
Dawn Song

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2013-85
http://www.eecs.berkeley.edu/Pubs/TechRpts/2013/EECS-2013-85.html

May 16, 2013

Transformation-aware Exploit Generation using a HI-CFG

Dan Caselden*, Alex Bazhanyuk*, Mathias Payer*, László Szekeres†, Stephen McCamant‡, and Dawn Song*
*Computer Science Division, University of California, Berkeley CA USA
Email: {dancaselden,virvdova}@gmail.com, mathias.payer@nebelwelt.net, dawnsong@cs.berkeley.edu
†Department of Computer Science, Stony Brook University, Stony Brook NY USA
Email: lszekeres@cs.stonybrook.edu
‡Department of Computer Science and Engineering, University of Minnesota, Minneapolis MN USA
Email: mccamant@cs.umn.edu

***Abstract*—A common task for security analysts is to determine whether potentially unsafe code constructs (as found by static analysis or code review) can be triggered by an attacker-controlled input to the program under analysis. We refer to this problem as proof-of-concept (POC) exploit generation. Exploit generation is challenging to automate because it requires precise reasoning across a large code base; in practice it is usually a manual task. An intuitive approach to exploit generation is to break down a program's relevant computation into a sequence of transformations that map an input value into the value that can trigger an exploit.**

We automate this intuition by describing an approach to discover the buffer structure (the chain of buffers used between transformations) of a program, and use this structure to construct an exploit input by inverting one transformation at a time. We propose a new program representation, a hybrid information- and control-flow graph (HI-CFG), and give algorithms to build a HI-CFG from instruction traces. We then describe how to guide program exploration using symbolic execution to efficiently search for transformation pre-images.

We implement our techniques in a tool that operates on applications in x86 binary form. In two case studies we discuss how our tool creates POC exploits for (i) a vulnerability in a PDF rendering library that is reachable through multiple different transformation stages and (ii) a vulnerability in the processing stage of a specific document format in AbiWord.

***Keywords*-Exploit generation; binary analysis; symbolic execution; data structure analysis**

I. Introduction

Security analysts currently spend much of their time analyzing software bugs to determine whether they might constitute vulnerabilities. This task currently has relatively little automated support, because it requires precise reasoning cross-cutting through a large code base; in practice it is still mostly manual. We propose new techniques to scale precise binary analysis to complex multi-stage information flows, in order to automatically produce test cases that demonstrate a vulnerability.

Specifically, we address the problem of *proof of concept (POC) exploit generation* for programs that perform multiple transformations of their input. Our system is given the location of a possibly unsafe code construct such as an instruction that might index an array out of bounds: for instance this might be the output of a conservative static analysis or from a core dump in a bug report. Our goal is to automatically produce an input to the program that demonstrates that the potential vulnerability is real. When the program executes on this new input, it will trigger the unsafe condition such as by accessing beyond the array bounds and crashing.

Exploit generation is a challenging task for several reasons. First, it requires reasoning about a program from the original program input to the potentially vulnerable code location. The data may undergo a number of different transformations between the input and the vulnerability, and the exploit generation process must take account of each. Second, exploit generation also requires precise reasoning since we require a specific input that demonstrates a vulnerability. It is not enough to determine that a vulnerability might be possible: a system must show how it really happens. (For this reason, we might also refer to the exploit generation task as *vulnerability verification*.) A third challenge is that we assume that we have only access to a compiled binary. Our technique does not depend on access to source code for the program, which may be unavailable to a third-party analyst, or for any libraries used whose source may not be available even to the application developer. Thus an exploit generation system should be able to work with *stripped binaries* that lack symbol tables or debugging information.

In part because of the difficulty of exploit generation (manual or automatic), many searches for security vulnerabilities use techniques such as fuzz testing that generate concrete crashing inputs directly. However concrete fuzz testing has many limitations of its own: it can require significant target-specific setup to perform well; it is unlikely to trigger vulnerabilities that are guarded by complex, low-probability conditions; and its results do not easily generalize. Better automated exploit generation is thus complementary to fuzz testing: it can produce exploits for some vulnerabilities that fuzz testing would take too long to discover, and if a vulnerability is initially found via fuzz testing, exploit generation can be used to create further exploits in new contexts or subject to additional constraints.

From an attacker's perspective, this capability for generating new exploits is particularly useful in constructing attacks that bypass signature-based filters. File formats that include complex transformations can be an easy way to implement sophisticated polymorphism for attacks on a single vulnerability. For instance in the context of PDF documents, we show in our case study how to exploit a single vulnerability in font parsing using font streams transformed via encryption, various forms of compression, and/or hexadecimal coding. Any one of these transformations on its own would defeat a simple signature-based attack detector, but our approach also extends to arbitrary sequences of transformations. If attackers have this capability, then no defensive filter can be effective unless it implements all the transformations that the vulnerable file-reading application (e.g., Adobe Acrobat) does.

When an analyst is faced with the manual exploit generation problem, a natural task is to work backwards through the program: for instance, first find an argument to the vulnerable function which causes a failure, then find the value for a previous variable that causes that argument, and eventually work back to the program input. Our approach is a way of automating this intuition. We first use binary analysis to build a program representation, what we call a Hybrid Information- and Control-Flow Graph (HI-CFG), which shows the data structures within a program as well as the code that generates and uses them. Based on the HI-CFG, we identify a sequence of data structures, which we refer to as *buffers*, that hold the data values leading to a potential vulnerability between a sequence of transformations. Then, we use an exploration technique based on symbolic execution to work backwards through this sequence of buffers. Our tool first finds contents for the final buffer which cause a failure, then finds contents for the second-to-last buffer which lead to those final buffer contents, and so on back to the original program input.

We implement this approach using binary trace collection, a dynamic HI-CFG construction system, and a symbolic exploration tool which is guided by the sequence of buffers found in the HI-CFG. In two case studies we generate POC exploits for (i) a font-related vulnerability in the Poppler PDF parsing library, which includes a complex decompression stage followed by an integer overflow within a textual header, and (ii) an illegal memory access when parsing an Office Open XML (`.docx`) file in AbiWord.

Our work makes the following major contributions:

- We apply a transformation-aware input generation approach to the problem of POC exploit generation. By taking advantage of the structure of buffers and transformations in a program, this approach lets our tool find exploits that could not be found by analyzing a program as a monolithic whole.
- We introduce a new program representation that integrates control flow and data structures, the Hybrid Information- and Control-Flow Graph (HI-CFG), and give algorithms for building a HI-CFG from instruction traces without source-level information.
- We show that the HI-CFG makes transformation-aware input generation possible by automatically determining a sequence of buffers and the transformations between them that connect the attacker-controlled program input to a vulnerable location.
- We present two case studies of real-world large programs: Poppler and AbiWord. In the case studies we use our tool to automatically generate POC exploits for bugs that are hidden behind several transformations.

The rest of this paper is organized as follows. Section II gives a more detailed overview of the problem and our approach to it. Section III describes our proposed Hybrid Information- and Control-Flow Graph (HI-CFG) representation, and the specific variant we use for exploit generation. Section IV gives the algorithms we use for HI-CFG construction, Section V describes how we use symbolic execution to invert single transformations, and Section VI describes how to use the HI-CFG to choose the sequence of transformations to invert. Section VII describes the case study in which we apply our system to real-world vulnerabilities, and Section VIII provides some additional discussion of our approach. Finally Section IX describes related work, and Section X concludes.

II. Overview

The intuition behind our approach is to automatically generate POC exploits by reversing buffer transformations in a binary-only program. The input to our system is (i) a potentially unsafe program location consisting of an instruction location and a vulnerability condition (a boolean formula) and (ii) a benign input that leads to an execution trace that executes the vulnerable functionality but does not trigger the vulnerability condition. Our tool builds a HI-CFG from the execution trace and uses this HI-CFG to identify buffers and to build a transformation chain between the input buffer and the faulting instruction. We then use symbolic execution to reverse individual transformations. The HI-CFG information about the transformation chain is then used to connect the inverse transformations into a POC generation toolkit. Figure 1 gives a high-level overview of the workflow.

We next turn to describing the technical structure of our problem and approach in more detail.

A. Problem

We refer to the problem our system addresses as *proof of concept (POC) exploit generation*. Given a potentially vulnerable location within a program, our goal is to produce a program input demonstrating an unsafe behavior at that location by searching executions similar to a given benign execution.

Vulnerability Condition. Our approach is based on analyzing the program at the binary level, so we represent a potential vulnerability as the address of an instruction

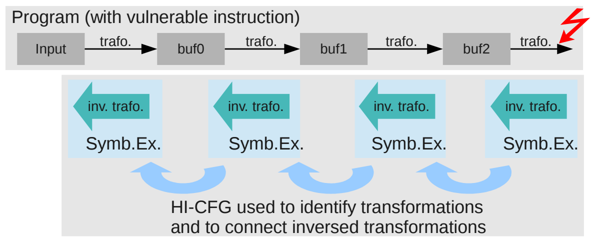


Figure 1. High-level overview of the workflow when generating POC exploits.

within a binary and a *vulnerability condition* on the state of execution at that point. The vulnerability condition is a formula in terms of registers and memory locations that, if true, indicates that the program will perform an unsafe operation at that instruction. For instance, if the vulnerability is a potential overflow of a 100-byte buffer indexed by the register `edx`, the vulnerability condition would be `edx` $\geq_u 100$, where $\geq_u$ indicates an unsigned comparison. Such a vulnerability condition might come from a conservative static analysis of a binary, which checks whether each instruction that accesses a data structure will respect its boundaries. Another possibility is for the vulnerability condition to represent a failure previously seen in a bug report: for instance, if we have a core dump caused by a null dereference at a particular instruction, we can create a vulnerability condition checking whether the memory address can be null. An instruction for which this conservative safety check fails represents a potential vulnerability, but a conservative analysis can produce false positives, so we can use exploit generation to find warnings that correspond to true positives. We can also find potentially vulnerable instructions in the course of a manual binary-level security audit, or translate them from source level conditions expressed in an `assert` statement. Our tool supports vulnerability conditions that are arbitrary formulas that combine the contents of registers and memory with fixed-size arithmetic, bitwise and logical operators.

Our primary focus in this work is to show that an unsafe operation can occur using a computed input, not to create a complete attack. The creation of a typical control-flow hijacking exploit can be broken down to four steps: (1) finding a potential bug that causes memory corruption; (2) constructing an input that will trigger the memory corruption; (3) tweaking the input so that a function pointer or other value in memory gets overwritten to eventually influence the argument of a control-flow instruction; (4) “weaponizing” the input with shellcode or with return-oriented programming [1] and “hardening” the attack to bypass common protections like DEP and ASLR.

Our technique focuses on step (2), what we refer to as proof-of-concept (POC) exploit generation. While we refer to what our tool does as *exploit generation* for brevity, we do not discount the importance of the other steps in the process. Because our tool is flexible about the format of the vulnerability condition, we can also express more specific necessary conditions for a vulnerability to be exploitable and cover step (3). For instance, extending the buffer overflow example above, we could ask for the overflowing value to be a particular negative number that causes a function pointer elsewhere in the machine state to be overwritten. Of course, more specific exploit details like this depend on the program’s memory layout, so they can only be checked at the binary level. This condition-based approach is also compatible with techniques for automated exploit hardening [2]. Furthermore, even if a potential vulnerability is found by fuzzing, meaning that an input is given which triggers memory corruption at a specific point in the program, our technique can still be used to decide whether the found bug is exploitable for control-flow hijacking attacks (by using additional conditions in the vulnerability condition).

Benign Input. The other input to our technique is a benign program input that leads to an execution similar to the desired exploit without triggering the vulnerability condition. Our tool uses this execution as the starting point for its search for an exploit. Because there will typically be many program inputs that could trigger a vulnerability, an analyst can vary aspects such as the size and structure of the benign input to affect the corresponding aspects of the generated exploit.

However, to make the search for an exploit efficient, the benign input should exercise code *near* the potential vulnerability. We do not require that the benign input execute the vulnerable instruction, but it should execute the function that contains the vulnerability. For instance, in the Poppler case study of Section VII, to exploit a vulnerability in the parsing of a Type 1 font in a PDF document, we choose as a benign input a document that includes a Type 1 font.

B. Approach

The intuition behind our approach is the observation that, in a large program, a value often undergoes a series of transformations between being read as input and triggering a potential vulnerability. Because of these levels of processing, it would be infeasible for our system to reason automatically about the entire transformation at once. Instead, our tool analyzes the static binary and an execution trace to understand the entire transformation as the composition of a series of simpler transformations.

Our tool recognizes the structure of composed transformations by the structure of the program’s control flow, its data structures, and how the code uses the data. Specifically, the analysis breaks the transformation down into an alternating series of buffers and smaller transformations. Each smaller transformation corresponds to a code module such as a function and the functions it transitively calls. A *buffer* is a data structure that is used as the interface between different parts of the program: the buffer contains data values that are generated by one part of the program and used by another.

Given this breakdown into a sequence of smaller transformations, our system then uses the breakdown to structure its search for a vulnerability-triggering input. Specifically, our analysis generates vulnerability-triggering values one buffer at a time. The analysis starts by searching for contents of the final buffer in the sequence that cause the program to fail (i.e., perform an unsafe operation). Assuming one is found, the analysis then searches for contents of the second-to-last buffer, which lead the program to produce the vulnerability-triggering contents in the final buffer. The analysis repeats this process for each of the remaining pairs of buffers until the program's input buffer is reached. If the system succeeds at each step, the final result is an input to the entire program that triggers the vulnerability.

A transformation-by-transformation approach can be much more effective than a search that treats the program as a single unit because there are many execution paths through each transformation. A transformation-by-transformation approach has a *search space that is the sum of the search spaces for each transformation*, while for a whole-program approach the search space size grows as a product of the individual search spaces (leading to a state explosion).

Our representation of the program and its buffers is a Hybrid Information- and Control-Flow Graph (HI-CFG). The HI-CFG is a graph that includes information about code, data, and the relations between them; we describe its structure in more detail in Section III. One could potentially use a number of kinds of program analysis to produce a HI-CFG, but for this project we use a dynamic approach that builds a HI-CFG representing the code and data structures exercised in one or more instruction-level traces. The details of this construction process are in Section IV.

Given the HI-CFG, our approach then uses symbolic exploration for the task of finding buffer contents that lead to a failure or to the desired failure-triggering contents of a later buffer. We refer to the latter process as finding a *preimage* of the desired output for the transformation. Symbolic exploration generates feasible execution paths of a portion of the program by constructing symbolic expressions for output values and path conditions, and solving those formulas using a decision procedure. We describe the basics of this technique, as well as optimizations that apply to the preimage-computation problem, in Section V.

In order to discover the buffers in a program and efficiently find preimages for the transformations between them, our system depends on three properties of the buffers and transformations, which we have found to be quite commonplace:

1) The buffers that our system detects must be stored consecutively in memory; this generally corresponds to data structures that are implemented using arrays in C or C++. Though not a fundamental limitation, we have focused on these data structures in our HI-CFG construction algorithm because they are the most common way for programs to store large quantities of data of a variety of formats.
2) The individual transformations should be mostly surjective: in other words, for any given transformation output, a corresponding preimage should exist. This property ensures that the sequence of preimage searches will not have to backtrack: when it finds buffer contents that produce the desired result for one stage of processing, it is likely that a sequence of further suitable preimages will exist back to the program input.
3) The individual transformations should be sequential and streaming: in other words, they should read from their input in order, write to their output in order, and the reads and writes should be intermixed.

These properties bound the size of the search space that must be explored to find a preimage: the search can reject a candidate input prefix when it leads to an incorrect output prefix. Examples of transformations that satisfy these properties include many kinds of decompression, character set transformations of text, signal processing, and encryption and decryption (though there can be other challenges in inverting cryptographic functions say if the key is not available).

III. The Hybrid Information- and Control-Flow Graph

For the central program representation used in our approach we propose what we call a Hybrid Information- and Control-Flow Graph ("HI-CFG" for short, pronounced "high-C-F-G"). The HI-CFG combines information about code, data, and the relationships between them. Because data structures represent the interface between code modules, a HI-CFG is a suitable representation for many tasks that require decomposing a large binary program into components. Listing 1 shows a simple transformation that copies data from buffer `buf0` to `buf1`. Figure 2 shows the HI-CFG that contains the control flow graph as well as the data flow graph and the producer/consumer edges between the two graphs.

We start by describing the kinds of nodes and edges found in a HI-CFG (Section III-A). Then we mention potential variations of the concept and applications for which they would be suitable (Section III-B). Finally, we describe the particular kind of HI-CFG we use for our exploit generation task (Section III-C).

A. Nodes and Edges

A HI-CFG is a graph with two kinds of nodes: ones representing the program's data structures, and ones representing its code blocks. Data structure nodes are connected with *information-flow* edges showing how information is

```
1  // ISO-8859-1 to UTF-8 conversion
2  void trafo(char *src, char *dst, int len) {
3    while (len-- > 0) {
4      if (*src < 0x80) {
5        *dst++ = *src++;
6      } else {
7        *dst++ = 0xc0 | (*src & 0xc0) >> 6;
8        *dst++ = 0x80 | (*src++ & 0x3f);
9      }
10   }
11 }
12 ...
13 trafo(buf0, buf1, 256);
```

Listing 1. A simple example transformation involving two buffers.

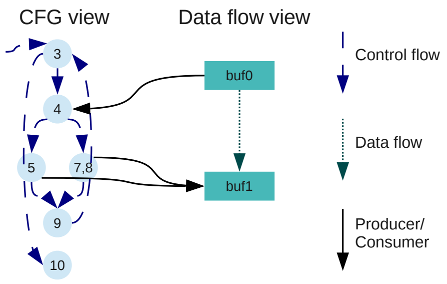


Figure 2. HI-CFG of the simple transformation in Listing 1 (the CFG nodes show the corresponding source code line numbers).

transferred from one data structure to another. Code block nodes are connected with *control-flow* edges indicating the order in which code executes. Finally, data nodes and code nodes are connected by *producer-consumer* edges, showing which information is created and used by which code: a producer edge connects a code block to a data structure it generates, while a consumer edge connects a data structure to a code block that uses it. A more detailed example HI-CFG is shown in Figure 3.

The subgraph of a HI-CFG consisting of code blocks and control-flow edges is similar to a control-flow graph or call graph, and the subgraph consisting of data structure nodes and information-flow edges is similar to a data-flow

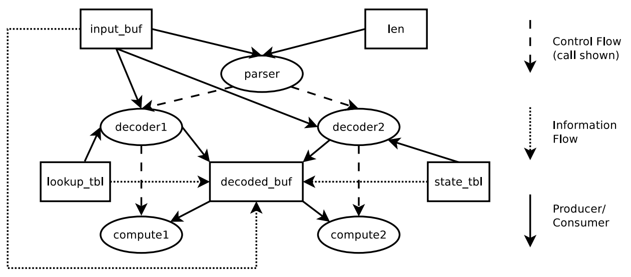


Figure 3. A detailed example of a coarse-granularity HI-CFG for a program which parses two kinds commands from its input, decodes those commands with the help of lookup tables, and then performs an appropriate computation for each command.

graph. However, the HI-CFG is more powerful than a simple combination of a control-flow graph and a data-flow graph, because the producer-consumer edges additionally allow an analysis to find the code that is relevant to data or the data that is relevant to part of the code.

A HI-CFG is distinguished from a program dependence graph (PDG), as used in slicing [3], [4], by its explicit representation of data structures. The nodes in a PDG represent code blocks, and two code blocks are connected by a data-dependence edge if one code block can read data written by the other. But the structure of the data values themselves is not represented in a PDG; it is only implicit in the placement of the data-dependence edges.

B. Generality and Uses

We can create a HI-CFG at differing levels of granularity for code and data. A fine-grained code representation would have one code block per instruction, or per basic block, while a coarse-grained representation would have one code block per function. Analogously, a fine-grained data representation would have a data structure node for each atomic value (like an integer), while a coarse-grained data representation would have one data structure node per allocated memory region. To record information about finer-grained structure, we can augment a coarse-grained data structure node with an inferred type that describes its internal structure.

When an analysis can recover only part of the information about a program's structure, such as when combining static and dynamic approaches, we can also annotate each HI-CFG edge with a *confidence* value between 0 and 1. A confidence value of 1 represents a relationship that our system knows definitively to hold, whereas a fractional value indicates an uncertain relationship, with lower values being less certain.

Component Identification. One application of a HI-CFG would be to identify functional components within a binary. The hierarchical, modular structure of a program is important at the source level for both developer understanding and separate compilation, but this structure is lost after a compiler produces a binary. Below the level of a dynamically linked library, a text segment is an undifferentiated sequence of instructions. However we would often like to determine which parts of a binary implement a certain functionality, such as to extract and reuse that functionality in another application. Caballero et al. [5] demonstrate the security applications of such a capability for single functions, but many larger functional components would also be valuable to extract.

An insight that motivates the use of a HI-CFG for this problem is that the connection between different areas of functionality in code are data structures. A data structure that is written by one part of the code and read by another represents the interface between them. Thus locating these data structures and dividing the code between them is the key to finding functional components. Given a HI-CFG, the functional structure of the program is just a hierarchical decomposition of the HI-CFG into connected subgraphs, and data structures connected to multiple areas of functionality represent the interfaces of those components.

Information-flow Isolation. A different kind of decomposition would be valuable for large programs that operate on sensitive data. In a monolithic binary program, a vulnerability anywhere might allow an attacker to access any information in the program's address space. But often only a small portion of a large application needs to access sensitive information directly. Just as automatic privilege separation [6] partitions a program to minimize the portion that requires operating system privileges, we would like to partition a program to minimize the portion that requires access to sensitive information. This problem can again be seen as finding a structure within the HI-CFG, but for information-flow isolation we wish to find a partition into exactly two components, where there is information flow from the non-sensitive component to the sensitive one but not vice-versa.

C. Application to Exploit Generation

For the purposes of this paper, of course, our application of the HI-CFG is to find the structure of a program's buffer usage to facilitate efficient exploit generation. For this, we use a relatively coarse-grained HI-CFG. We represent code at the level of functions, so control-flow edges correspond to function calls and returns. To represent data structures, we use a level of granularity intermediate between atomic values and memory allocations: our tool detects buffers consisting of adjacent memory locations that are accessed in a uniform way, for instance an array. Our current prototype implementation detects only one level of buffers, so we do not infer types to represent their internal structure.

Because our HI-CFG construction algorithm, as described in Section IV, is based on dynamic analysis, each edge in the HI-CFG represents a relationship that was observed on a real program execution. Thus all edges effectively have confidence 1.0. The converse feature of this dynamic approach is that relationships that did not occur in the observed execution do not appear in the HI-CFG. However this is acceptable for our purposes because we base the HI-CFG, and thus the search for an exploit, on an analyst-chosen benign execution. If the first benign input does not allow our tool to find an exploit, the analyst can try again with a benign input that exercises different parts of the program functionality.

We will return to the question of how to find buffers and transformations in the HI-CFG in Section VI, after covering the building blocks of how to construct the HI-CFG (Section IV), and how to find transformation preimages (Section V).

IV. Dynamic HI-CFG Construction

In this section, we describe our approach to HI-CFG construction: first some infrastructure details, then techniques fo collecting control-flow information from dynamic traces, categorizing memory accesses into an active memory model, grouping data accesses into buffers, tracking information flow via targeted taint analysis, and merging significantly similar buffers.

A. Infrastructure

To construct a HI-CFG via dynamic analysis, we take a trace-based approach, which lets us easily explore and test a variety of specific techniques and evaluate them with concrete examples.

We use the open-source Tracecap [7] tool from UC Berkeley to record instruction traces. Tracecap records statistics about loaded executables and libraries, tracks the entry of tainted data to the process space, and produces a log of function calls including arguments and return values that we later use to track standard memory allocation routines.

Our modular trace analysis system interfaces with Intel's XED2 [8] library (for instruction decoding) and Tracecap (for reading and writing instruction traces). It includes an offline taint propagation module that allows for a virtually unlimited number of taint marks, and a configurable number of taint marks per byte in memory and registers.

B. Control Flow

The HI-CFG construction module primarily identifies functions by observing `call` and `ret` in the instruction trace. Upon observing a `call` instruction, the module will update the call stack for the current thread and create a control-flow edge from the caller to the callee. Upon observing a `ret` instruction, the module finds the matching entry in the call stack and marks any missed call stack entries as invalidated.

In addition to literal `call` instructions, our system also recognizes optimized tail-calls by noticing execution at addresses that have otherwise been `call` targets. A limitation of this approach is that tail-called functions will never be recognized if not directly called. This limitation of the current implementation could be addressed by adding a static analysis step to the HI-CFG construction process, but it has not been a problem so far.

C. Memory Hierarchy

The HI-CFG construction records memory accesses in a hierarchical model of memory which follows the lattice shown in Figure 4. `space` types at the top of the lattice represent an entire process address space. At the bottom of the lattice, `primitives` represent memory accesses observed in the instruction trace. The categorization of a memory access corresponds to a path from the top of the lattice to the bottom. Existing entries in the

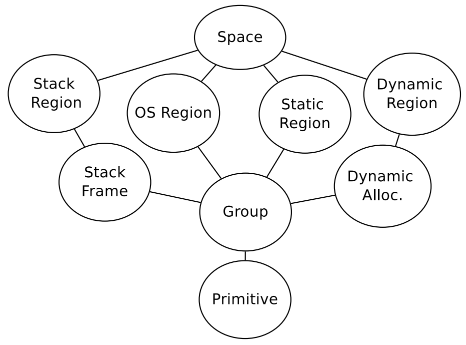
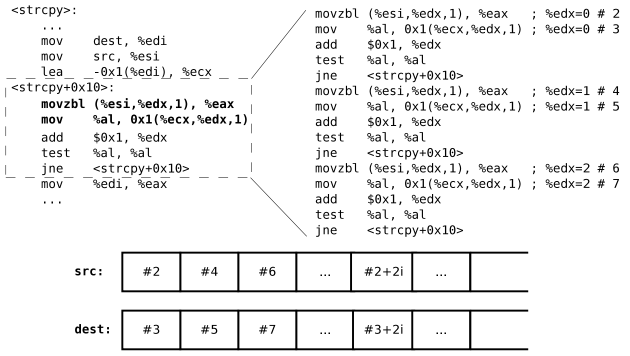


Figure 4. The hierarchy of types in the model of memory used in our HI-CFG construction algorithm.



```
<strcpy>:
    ...
    mov    dest, %edi
    mov    src, %esi
    lea    -0x1(%edi), %ecx
<strcpy+0x10>:
    movzbl (%esi,%edx,1), %eax
    mov    %al, 0x1(%ecx,%edx,1)
    add    $0x1, %edx
    test   %al, %al
    jne    <strcpy+0x10>
    mov    %edi, %eax
    ...
```

```
movzbl (%esi,%edx,1), %eax   ; %edx=0 # 2
mov    %al, 0x1(%ecx,%edx,1) ; %edx=0 # 3
add    $0x1, %edx
test   %al, %al
jne    <strcpy+0x10>
movzbl (%esi,%edx,1), %eax   ; %edx=1 # 4
mov    %al, 0x1(%ecx,%edx,1) ; %edx=1 # 5
add    $0x1, %edx
test   %al, %al
jne    <strcpy+0x10>
movzbl (%esi,%edx,1), %eax   ; %edx=2 # 6
mov    %al, 0x1(%ecx,%edx,1) ; %edx=2 # 7
add    $0x1, %edx
test   %al, %al
jne    <strcpy+0x10>
```

Figure 5. Identifying spatially adjacent seqences of memory accesses on a trace of `strcpy`.

memory model add their own types as additional requirements in the path. For example, a memory access under an existing dynamically allocated memory region will at least have the path `space`, `dynamic region`, `dynamic allocation`, `primitive`. The memory model will then insert the memory access and create or adjust layers according to the types in the path.

Memory structures such as dynamic allocations and stack frames are added to the memory model as they are identified by one of several indicators. Dynamic allocations are added to the memory model by tracking standard memory allocation routines such as `malloc` and `free`. Stack frames are created by tracking `esp` during `call` instructions and claiming all memory accesses between the base of the stack frame and the end of the stack region during matched `ret` instructions.

Memory structures such as stack and dynamic regions are based on memory pages. The "region" type classification relies on the intuition that most programs tend to use each page for a single purpose such as for stacks, dynamic allocations, memory-mapped executables, or operating system structures. In addition to the constraints implied by the lattice, additional constraints prohibit stack frames and dynamic allocations from appearing in the memory model without their respective regions.

D. Grouping Buffers

Instruction traces contain every individual load and store instruction performed by the traced program, but for the HI-CFG we wish to group these accesses into buffers to better understand their structure. We identify buffers as groups of adjacent memory locations between which the program expresses commonality.

We experimented with several heuristics for identifying buffers and currently use a combination of two approaches. Our first system recognizes instructions that calculate memory access addresses by adding an index to a base pointer. The system searches the operands involved in the address calculation for a suitable base pointer (which must point to an active page of memory). Upon finding a suitable base pointer, the system submits a candidate buffer consisting of an address equal to the value of the base pointer and a size that extends the buffer from the base pointer to the end of the observed memory access. For example, analyzing a one-byte memory access of address `0x800000ff` by the instruction `movzbl (%esi,%edx,1), %eax` where the base pointer `esi` is `0x80000000` would yield a `0x100`-byte candidate buffer from `0x80000000` to `0x800000ff`.

The first system often detects both arrays consisting of homogeneous data types and structures consisting of heterogeneous data types. However, it fails when the address of the memory access is constructed by pointer arithmetic across multiple instructions. Our second system addresses this weakness by recognizing spatially adjacent memory accesses. To reduce the false positive rate of buffer detections, our second system also tracks the order of memory accesses within each function. Upon observing a return instruction and updating the call stack, or freeing a chunk of dynamically allocated memory, the second system uses the accesses from the returned function or freed memory as starting points to search through the active memory model for linear access patterns. If found, the system will submit the candidate buffer for a final stage of processing.

The memory access patterns in case of `strcpy` can be seen on Figure 5. Similar access patterns across multiple calls to the same function, such as by functions that access one byte of a buffer per call, are also recognized by this system. In addition, access patterns are stored within buffers so that they may grow with subsequent accesses.

Once the first and second systems have submitted their candidate buffers, the HI-CFG module chooses the most suitable buffers (with a preference for larger buffers) and commits them to the active memory model. Adding a buffer to the active memory model merges the grouped memory accesses with the new buffer, which summarizes relational information such as producer and consumer relationships with functions and information flow to other buffers, which are described in the next subsection. Subsequent buffers submitted to the active memory model will merge their relational information with existing buffers, and potentially extend the buffer if certain criteria are met (e.g., the buffers share a starting or ending address).

E. *Information Flow*

To trace the information flow between buffers, our system primarily uses a specialized form of dynamic taint analysis. We introduce a fresh taint mark for each buffer as a possible source for information flow. We then propagate these taint marks forward through execution as the data values are copied into registers and memory locations. When a value with a taint mark is stored into another buffer distinct from the source buffer, we record an information flow from the source to that target. Like most techniques based on dynamic taint analysis, this technique will not in general account for all possible implicit flows, so we also supplement it with an upper-bound technique that constructs a low-confidence information-flow edge when ever the temporal sequence of buffers consumed and produced by a function would allow an information flow.

F. *Buffer Summarization*

Buffers in the active memory model are moved into the historical memory model when they or their hierarchical parents are deactivated. Primarily, this occurs for stack allocated buffers (when functions return) and dynamically allocated buffers (when the allocated chunk is freed). The remaining entries in the active memory model are deactivated when the HI-CFG construction module analyzes the last instruction in the trace.

Passthrough buffers, through which information flows without being acted upon by multiple functions, are not added to the historical memory model after deactivation. The motivations for this phase are twofold: first, passthrough buffers are generally less interesting for our analysis and their removal is a slight optimization; second, passthrough buffers will connect legitimately separate sections of the HI-CFG with information flow. Removing passthrough buffers improves the precision of the HI-CFG by eliminating cases that would indicate spurious information flow: for instance, if `memcpy` copied through an internal buffer that were not removed, every source of a copy would appear information-flow connected to every target.

We define passthrough buffers as those that satisfy the following criteria:

- The buffer is not a source of information flow (i.e., it has at least one incoming information flow edge).
- The buffer is not a sink of information flow (i.e., it has at least one outgoing information flow edge).
- The buffer is produced and/or consumed by exactly one function.

If all of the criteria are met, the passthrough buffer is removed from the graph, and new information flow edges connect buffers that were connected by the passthrough buffer. When deactivated buffers do not meet the criteria for passthrough buffers, they are moved into the historical memory model and *summarized*, as we describe next.

The summarization process finds buffers that are related (intuitively, multiple instances of the "same" buffer), and merges them along with their relational information. We define when two buffers should be merged by giving each buffer a value we call a *key*. Two buffers should be merged if they have both the same parent (perhaps because previously-separate parents were themselves merged) and the same key. (To save storage space, we in fact just store an MD5 hash of the key material.) The key always includes an identifier for the type of an object, and by default it also contains the object's offset within its parent object.

The keys for dynamic allocations and stack frames contain different information in addition to a type identifier. Dynamic allocations use the calling context of the allocation site, up to a configurable depth (currently set to 10 calls), similar to a probabilistic calling context [9]. Stack frames use the address of the function. As a result, our system is able to identify two local and dynamic variables as the same across multiple calls to the same function and in the presence of custom memory allocation wrappers.

We use an approach similar to a disjoint-set union-find data structure to manage the identities of buffers as they are summarized. The merging of buffers corresponds to a *union* operation, and we use a *find* operation with path compression to maintain a canonical representative, associated for instance with a taint mark. This allows the tool to efficiently maintain information-flow from historical buffers even after they are deactivated.

V. Pre-image Search via Symbolic Execution

Our exploit-generation system uses symbolic execution to search for buffer contents that either trigger a vulnerability, or lead to a desired value being generated in a subsequent buffer. Except for the difference in goal, the two search processes operate in a similar way. We start by giving a brief introduction to the use of symbolic execution for program exploration (Section V-A), then describe three features and optimizations that make the preimage search efficient: pruning (Section V-B), prioritization (Section V-C), and the treatment of lookup tables (Section V-D).

A. *Background: Symbolic Exploration*

Symbolic execution is a program analysis technique that combines features of dynamic and static analysis by considering families of executions that traverse the same execution path. Certain inputs to a program or code fragment under test, rather than taking concrete values such as particular integers, are replaced by symbolic variables. As the code executes, computations on these values produce more complex symbolic expressions. When a symbolic expression occurs in a branch condition, we can use a decision procedure such as STP [10] to determine which directions for future execution are feasible. Symbolic execution is useful because the symbolic execution of a single path can correspond to a large number of concrete executions, but still be precise: no approximation is involved in computing the symbolic expressions. Another advantage is that arbitrary additional conditions can be conjoined with the formulas (as if they were additional branches in the program) and checked in the same way. For instance, in this way our tool can easily check whether a vulnerability condition is satisfiable.

One common application of symbolic execution is to explore within the space of all feasible executions of a code fragment, which we refer to as *symbolic exploration*. Our system takes an approach that explores one execution path at a time, starting with no constraints on the symbolic variables. Each time execution reaches a symbolic branch, the tool is free to explore either side of the branch, subject to a feasibility check (ensuring the choice is compatible with the earlier branches taken). The tool keeps track of which sequences of branch choices lead to parts of the execution space that have been fully explored, and avoids them. We describe a further basis it uses for deciding between branch directions below in Section V-C; when all else is equal, it chooses randomly. A more in-depth discussion of symbolic execution techniques is found in the form of a survey [11], or in papers describing tools [12], [13]. Our implementation builds on an existing tool [14].

If it were allowed to run forever, a symbolic exploration tool would eventually explore every possible execution path through a code fragment. But for all but the smallest fragments, the number of paths is so large (even infinite) that this is not a practical strategy. The key to effective use of symbolic exploration is to guide the search towards execution paths that are more likely to be interesting, which will be the focus of the next two subsections.

B. Search Pruning

The most important technique for reducing the size of the space that must be searched for a preimage is to prune prefixes of the input buffer contents that produce the wrong prefix of the output buffer contents. This technique applies to streaming transformations that read their input and write their output sequentially, and also closely interleave writes with reads. This is a very common pattern for transformations that can apply to an unbounded input, but keep only a limited-size internal state.

While exploring the execution of the transformation, at each point at which the code writes a value to the output buffer, we check whether it is possible that the written value can be equal to the desired output value at that position. If it cannot be equal, then no extension of the currently explored path could create the desired output, so the search can be pruned at that point, and no extensions are explored. If the values can be equal but are not necessarily equal, such as if the written value is an unconstrained symbolic variable, we add the constraint that they match to the path condition, which can also prune the search space by making some future paths infeasible.

An indication of the power of this pruning is that if the number of reads between consecutive writes is bounded, it will typically reduce the number of paths that can be explored from exponential in the input size to linear in the input size. However applying just this technique the linear factor can still be quite large, which can be further addressed by the two optimizations we describe next.

C. Search Prioritization

Another optimization that can take advantage of checking as the code produces the desired output is to bias the search toward paths that have produced the most correct output values. Intuitively, this approach directs the exploration to spend more of its time attempting to extend paths that have already proved promising, as opposed to paths that have not shown results yet. This approach can be described in terms of a utility function for states in the exploration space, which for our preimage computation is the number of correct output values produced by the path up to that point. To implement this approach, our tool records, before each branch point in the search space, the minimum and maximum utilities of all of the states that have been explored beyond that point. When returning to a branch point that has been visited before, the search will prefer the branch direction with the higher maximum utility, or if the maximum utilities are equal it will prefer the one with the higher minimum utility.

As in any search process, our search for a preimage has a tension between local and global search. Prioritizing states that have proved effective so far will speed the search if the search space is well behaved. But one would not want to unconditionally prefer the already-proven states, because the search space might have dead ends that appear initially promising, but cannot be extended to give the complete desired output. We do not want a search process that is required to explore such dead ends exhaustively before trying another path. To strike this balance, our system's search prioritization is not absolute. Instead, each time the search reaches a state with a utility-based preference, we flip a biased coin. If the coin comes up heads, we follow the preference, otherwise we fall back to a random choice strategy. For the experiments in this paper, we have set the probability of the biased coin to follow the utility-based preference with probability 0.95. This probability works well for our case studies and follows a greedy strategy that ensures that we first search the depth of the tree before backing up and searching more in the breadth.

D. Lookup Tables

A final aspect of our use of symbolic execution is not specific to exploring transformations, though it often applies to them. As previously mentioned, an advantage of symbolic execution is that a single symbolic path can correspond to multiple concrete paths. A trade-off with respect to how many concrete paths a symbolic path represents occurs when the code uses a load from memory to implement a lookup table.

This trade-off arises when the address value used in a load from memory is symbolic, so that the load might refer to multiple locations. How should the symbolic execution system implement this load?

One approach, which is the default in our symbolic execution tool, is to treat the selection of which address to load like a multi-way branch. The tool chooses one feasible value for the address, and continues execution subject to this choice. Later, when it returns to the branch point, it can choose a different feasible value. Since choosing a value for the address effectively makes it concrete, this approach tends to create simple symbolic formulas which can be evaluated efficiently. On the other hand, if many address values are possible, the number of paths to explore can quickly become large.

An alternative approach is to represent all the possible values from the load in the symbolic expression. This approach makes sense for the case in which the possible loaded values represent a lookup table, though it need not be limited to that case. The formula representing the symbolic results of the load will itself have the structure of a lookup table (or, equivalently, a circuit representing a ROM). The main limitation is that the number of possible addresses and loaded values cannot be too large, lest the symbolic formula become unmanageable.

The trade-off that comes with the lookup table approach is that the number of execution paths to be explored will be much smaller, but at the cost of the symbolic formulas for each path becoming much larger and slower to reason about. Essentially this approach delegates more of the exploration to the decision procedure. It can improve performance overall because the decision procedure can use many of its own optimizations, though representing and reasoning about large formulas can increase memory usage.

The table lookup approach is perhaps more natural in source-level symbolic execution systems that know when a variable has array type. In binary-level symbolic execution, the first challenge is to recognize when a table lookup is occurring. Our system detects a table lookup when the effective address of a load is the sum of a constant value and a symbolic one, when the constant value is in the range of a memory address, and the symbolic value (treated as the table index) is bounded. Specifically we recognize table sizes that are a power of two, with smaller tables rounded up, which makes the construction of the lookup formula more convenient. In some cases the bound on the index expression is evident from its syntax (for instance, if it is zero-extended from a byte value); if not, our tool uses additional decision procedure queries in a binary search to find the smallest power-of-two bound. The maximum allowed table size is configurable; for this paper, it was 2^{16}.

VI. Choosing Buffers and Transformations

Next we describe how our system uses the information from the HI-CFG to determine the relevant buffers that lead to a potential vulnerability, and the transformations on which we apply symbolic exploration to find preimages.

A sequence of transformations leading to the function containing a potential vulnerability will appear in the HI-CFG as a path (as shown in the motivating example in Figure 1). The first node in the path is a buffer representing the program input. The remaining nodes in the path before the last are additional buffers internal to the program, connected by information-flow edges. Finally the path ends with a consumer edge leading to the function containing the potential vulnerability.

In general, the HI-CFG may contain multiple paths of the form described above. For instance, in addition to the buffers containing the primary data that the program is processing, there may be an additional information-flow path of buffers containing meta-data. In choosing a path to search for an exploit, we expect to have more luck with the primary data than with meta-data. To implement this preference, our system attempts to distinguish buffers that are more likely to contain primary data by their larger size. Among all the paths of the form described in the previous paragraph, we choose the path for which the size of the smallest buffer on the path is maximized.

Once the tool has chosen the sequence of buffers to trigger the vulnerability, the HI-CFG also contains information about which functions in the program implement the transformation from the contents of one buffer to the contents of another. Specifically, each function that implements part of the transformation will have a consumer edge from the earlier buffer, and a producer edge to the later buffer. In the case where the transformation is spread across multiple functions, the nearest call-graph ancestor that dominates all of the functions connected to both buffers will generally be a function whose execution performs the transformation.

VII. Case Studies

As case studies, we have applied our exploit-generation system to two applications that transform their inputs before processing them. These programs are open-source, and we use the source code to verify our results, but the system does not use the source code or source-level information such as debugging symbols.

A. *Poppler*

Poppler is a PDF processing library used in applications such as Evince. The vulnerability for which we generate an exploit is cataloged as CVE-2010-3704 [15]. PDF documents can contain embedded fonts in the Type 1 format, which is derived from PostScript and allows properties and the character encoding to be expressed in a flexible text format. Poppler includes a lightweight parser that attempts to recover the character encoding, a mapping from byte values to character glyphs. The index values for the encoding are intended to be integers between 0 and 255 inclusive, and the syntax does not allow a minus sign to indicate a negative decimal integer.

However the syntax does allow integers to be specified in octal with a prefix of `8#`, and Poppler's conversion from ASCII octal to binary does not check for overflow. Further, the check that determines whether the code value is in bounds uses a signed comparison to 256, so a negative value will be incorrectly accepted. Thus, if a malicious font specifies a very large octal value for a character position, it will overflow to a negative value. When this negative value is multiplied by 4 and used to index the encoding array, an arbitrary location elsewhere in memory will be overwritten.

The "stream" that contains an embedded font within a PDF document is typically compressed to save space; it can also be encrypted if the document uses access control, or transformed using a number of other filters. By applying our system with benign documents that use various filters, we can create PDF files where the exploit is transformed in various ways. For current versions of PDF the most commonly used compression format is named `FlateDecode`, an implementation of the Deflate algorithm specified in RFC 1951 [16]. We'll first describe how our technique works when the font is Flate-compressed, and then discuss other filters.

We apply our system to create a POC exploit for this vulnerability in a PDF document containing a Type 1 font. We use a test program `pdftoppm` included with the Poppler library that renders a PDF document to a bitmap. As the vulnerable instruction, we take the `jg` branch that performs a signed comparison between the index value and 256. A simple version of the vulnerability condition would simply require that the index value be negative.

Of course more steps are needed to weaponize the exploit, and especially to create an exploit that works in the presence of defense mechanisms such as ASLR and W$\oplus$X. We have confirmed that the vulnerability can be exploited in the presence of modern defenses by building by hand a complete exploit. The exploit overwrites the virtual destructor vtable entry of the font object (part of a nearby heap object unaffected by ASLR) with the entry point of ROP-based shellcode that calls the `execve` system call.

Completely automating the construction of such a hardened exploit is outside the scope of the present project, but has been the subject of other research such as the Q system [2]. Since our symbolic execution approach is similar to systems such as Q, we can show how to use our system with a more complex vulnerability condition as part of constructing such a hardened exploit. For instance consider the task of constructing an octal value which when used as an index in `%edx` will cause the font object vtable to be overwritten. The object pointer is at an offset of -316 from the frame pointer, and the vtable and the encoding array pointers within the object are at offsets 0 and 24 respectively. So the vulnerability condition is:

```
mem[mem[ebp-316]+24] + 4*edx == mem[ebp-316]
```

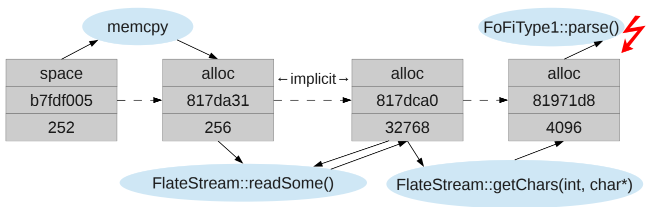


Figure 6. An excerpt of the HI-CFG for our Poppler case study showing the buffer sequence. The input travels from left to right and FoFiType1::parse contains the vulnerability.

As a benign input, we use a PDF file generated by `pdftex` applied to a small TEX file, which contains a FlateDecode-compressed Type 1 version of the Computer Modern Roman 10 point font.

An excerpt of the relevant portion of the HI-CFG generated by our tool from the processing of the benign input is shown in Figure 6. Input passes through a sequence of four buffers before the vulnerable code is triggered, so our tool must find contents for the final buffer which trigger a vulnerability in the font parser, and then it must compute three levels of preimages of this value. However, two of the transformations are direct copies for which preimage computation is trivial. The first and second buffers contain information in the same format. They both contain data directly from the input file: the first buffer is maintained by the standard I/O functions of the C library, while the second is a FileStream object that is part of Poppler. The third and fourth buffers also are in the same format: the third buffer is a FlateStream object containing a decompression buffer, and the fourth is a copy of the completely decompressed font. The non-trivial inversion between the second and third buffers computes a preimage under the FlateDecode transformation: a compressed font that decompresses to the attack font.

The execution trace from the benign execution contains 13,560,478 instructions, and constructing the HI-CFG took about 4.5 hours (16207.58 s) on a Xeon X5670. The generated HI-CFG contains 947 functions and 18654 groups.

Computation	Paths explored	Run time (s)
Font parse	36	107.57
Font copy	1	32.29
FlateDecode	111	6598.69
`fread` copy	1	16.99

Figure 7. Statistics for the symbolic exploration of four computations within the execution of the Poppler PDF parsing library, for generation of a POC exploit. Our tool generates a font that triggers a vulnerability within font parsing, and then computes preimages through two copy transformations and a decompression routine to create contents for the compressed PDF file to trigger the failure.

Statistics for the symbolic exploration are shown in Figure 7; running times are from an Intel Core 2 Duo E8400. Since the search process is randomized, the paths explored and the running time vary: we took 10 runs, dropped the fastest and slowest, and report the average of the remaining 8.

By comparison, applying the same symbolic exploration tool to the whole program at once, with the relevant part of the font stream symbolic and lookup tables treated symbolically, ran for more than 12 hours without even reaching the vulnerable instruction. In fact, this search was only able to explore 15 paths: with no pruning possible, each path became very long. Treating lookup tables concretely allowed the search to cover 898 paths in 12 hours, but still without reaching the vulnerable instruction.

To trigger the vulnerability in font parsing, our tool makes a 16-byte portion of the font symbolic, and searches for contents which trigger the vulnerability condition. We help the search by supplying 8 preferred directions for branches. These cause the tool to prefer executions in which three loops, which check for newlines, null characters, and the end of the sequence of octal characters, each do not end early. Similar annotations have been automatically generated by static analysis in previous work [17]; the loop exhaustion strategy of AEG [18] is also similar. The tool finds the contents "`␣8#0027777774674`" (where the first character is a space).

Even considering this stage on its own shows how this vulnerability could not feasibly be found by random fuzzing. Long octal numbers would be unlikely to appear in benign seed inputs, and if a tool were to choose characters at random, the probability of choosing a string of the form `8#oooooooooooo`, in which each character *o* is between `0` and `7`, would be about $4 \cdot 10^{-25}$. We confirmed this by applying a random fuzzing tool to modify the benign compressed data: the tool generated 63834 inputs in 12 hours, and more than 500,000 over about 4 days, without triggering the vulnerability.

The other non-trivial task for symbolic execution is to effectively find a sequence of bytes in the DEFLATE compressed data format which decompress to the vulnerability-triggering octal number mentioned above. The decompression routine uses several large tables, such as a table with 2^{14} entries to decode Huffman code words that, in this stream, can be up to 14 bits long. The tool treats each of the table lookups symbolically, which reduces the number of paths to explore at the cost of making each decision procedure query relatively expensive. On average this search takes a little less than 2 hours.

Another commonly used transformation of streams in PDF files in RC4 encryption. In the "user password" mode, PDF files are unreadable unless a required password is supplied to decrypt their streams. In "owner password" mode, information sufficient to recover the encryption key is provided inside the PDF, but DRM-compliant implementations will refuse to perform operations disallowed by a permission bitmap. It is easy for our tool to re-encrypt modified data by constructing pre-images for RC4 decryption because RC4 is a stream cipher, and the key is fixed. Essentially our system automatically implements the standard mutability attack against a stream cipher by solving constraints of the form "decrypted byte = encrypted byte XOR fixed keystream byte". We applied our technique to a version of the previously described sample document with RC4 and an owner password. Since no branching is involved, only one symbolic path needs to be explored, and the running time is 20 seconds, mostly devoted to program startup.

Two further transformations supported by PDFs include run-length encoding and a hexadecimal encoding of binary data. PDF's simple run-length encoding operates at the byte level: bytes indicate either a repeat count for a single byte, or a run of bytes to be copied to the output verbatim. The hexadecimal encoding is intended for representing binary data in printable ASCII: each byte is represented by two case-insensitive hex digits, and whitespace is skipped. We test inverting these two transformations with a PDF file that again contains the benign Type 1 font, but run-length encoded and then hex-encoded. These transformations are again relatively easy to invert via symbolic execution; we use two branch-direction annotations, like the ones described earlier for DEFLATE, to prefer hexadecimal strings that do not include whitespace. The preimage computation requires 143 seconds and 315 symbolic paths.

B. AbiWord

AbiWord is a word-processing application that can import documents from a number of formats. In particular we examined its processing of documents in Office Open XML format (used with the extension `.docx`), which is the default format of recent versions of Microsoft Word. An Office Open XML document is structured as a compressed Zip file containing multiple XML documents representing the document contents and metadata.

Recent versions of AbiWord (we used 2.8.2) suffer from a crash in XML processing that is triggered when a *shading* tag occurs outside of a *paragraph* tag. The code attempts to fetch the top element of an STL stack containing pointers to

```
<w:document xmlns:w="http://schemas.
openxmlformats.org/
wordprocessingml/2006/main">
<w:body><w:sdt><w:sdtEndPr>
<w:xxxxxxxx>
<w:shd w:fill="AAAAAAAAAAA"/>
</w:xxxxxxxx>
</w:sdtEndPr></w:sdt></w:body></w:document>
```

Figure 8. When the string "xxxxxxxx" in the above XML document is replaced with a certain tag name, the resulting `.docx` file crashes AbiWord version 2.8.2. Our symbolic execution tool automatically finds that "`rPr␣␣␣␣␣`" and "`pPr␣␣␣␣␣`" cause the crash.

enclosing document objects, but when the shading tag occurs in an unusual (but legal according to the schema) location, the stack is empty causing the reference to dereference an invalid pointer. We have not yet determined whether this bug is exploitable: a common presentation is a page-zero load (from an address of 504, computed as a null pointer, plus the length of an internal object in an STL deque, minus the size of one stack element).

The execution trace collected from the benign execution contains 69,503,117 instructions, and constructing the HI-CFG took about 9.3 hours (33557 s). The generated HI-CFG contains 5139 functions and 7816 groups. Looking at the sequence of buffers in the HI-CFG, the document data starts in a standard-IO input buffer, and is then decompressed by the `inflate` function. The decompressed buffer is then copied unchanged via `memmove` into a structure called the parser context, which is used by `xmlParseDocument`; the function containing the vulnerability is a callback from this parser.

As shown in the HI-CFG, there are two non-trivial processing steps between the input file and the crash: a decompression transformation from the Zip-file encapsulation, followed by a vulnerable XML parser. Our POC exploit generation process starts with the XML parser. Since the crash is caused by a legal but unusual nesting of XML tags, an exploit could be generated straightforwardly by a concrete fuzzer that was aware of the schema for `.docx` files. However building such a fuzzer requires significant effort that would have to be duplicated for each new file format fuzzed. By comparison symbolic execution is computationally more expensive, but its search effectively infers the grammar of the input file automatically, replacing human effort with machine effort. It is beyond the current capabilities of our symbolic execution tool to generate the crashing input from scratch, but we can demonstrate its ability to reason about possible inputs by using it to convert a benign file into an exploit. (It would also be possible to integrate a grammar-aware concrete fuzzer with the HI-CFG, something we leave for future work.)

The non-crashing XML document is shown in Figure 8. This document is not legal because `xxxxxxxx` is not a valid tag, but out of approximately 600 Office Open XML tags recognized by AbiWord's `.docx` import plugin, two will cause a crash when they are inserted in place of `xxxxxxxx`. To find these tags using our symbolic execution tool, we mark the `x` bytes as symbolic, and add constraints requiring that the bytes represent either letters or trailing spaces, and that the opening and closing tags match. Though the space of possible values for the "x"es is astronomical, even under the additional constraints (more than $52^8 > 4 \cdot 10^{11}$), the number of execution paths taken for invalid tags is relatively small, because AbiWord can determine relatively early that a tag is invalid. Specifically, AbiWord checks whether tags are valid by using an STL map (implemented as a red-black tree) and `strcmp`. Splitting the search to run in parallel across the 26 possible initial letters, our tool finds the exploit tag `rPr` after 53 symbolic paths and 2423 s in the "r" search, and `pPr` luckily after 3 iterations and 86 s in the "p" search. (This would roughly correspond to 17.5 hours of sequential search; speedup was less than linear because the Xeon X5670 used has only 12 cores.)

Given the crash-inducing XML text, our tool finishes the task of producing a exploit `.docx` file by finding a preimage for the compression used for the XML text in the `.docx` file's Zip encapsulation. In fact Zip files use the same DEFLATE algorithm mentioned earlier in the Poppler case study, though an independent implementation, so we do not repeat the details. The symbolic execution tool runs somewhat faster than in the Poppler example because the benign input file we generated used a smaller Huffman tree which required smaller decoding tables. On average (across 10 runs dropping the fastest and slowest), the search requires 237 seconds and 92 symbolic paths.

VIII. Discussion

Next we discuss in more detail some of the features, applicability, and limitations of our approach.

A. Sources of Vulnerability Conditions

Our tool attempts to construct an exploit for a given vulnerability condition; it does not itself search for a potential vulnerability. For instance, it can be used to verify or exploit potential vulnerabilities discovered via static analysis, dangerous situations discovered during manual code audit, a crash location recovered from a core dump file, or application-specific conditions expressed via assertions. If a POC exploit already exists, the tool could also be used to construct new exploits based on different benign inputs.

B. Invertible Transformations

Our approach for computing inverse images via symbolic execution depends on several features of a transformation implementation in order to find an inverse efficiently. While common, these features are not universal.

First, our tool is designed for transformations whose input and output come via contiguous data structures such as arrays that are accessed sequentially. With additional data-structure inference, the approach could be extended to more complex linked and nested structures. However, observe that for pruning to apply, it must be clear when the transformation has committed to an output value: our current approach works when each output location is written exactly once.

Second, pruning is most effective if the transformation's input and output are closely interleaved, so that unproductive paths can be pruned early. This is typical of streaming transformations that keep a small internal state. By contrast, if a transformation performs no output until the end of its execution, pruning will have no benefit.

One example of a class of transformations that do not satisfy these features, and cannot generally be inverted by our approach, are cryptographic hash functions. Of course, such functions are explicitly designed to be difficult to invert.

C. Multiple Threads

Our trace collection system and the HI-CFG construction algorithms are designed to accommodate traces of multi-threaded applications, but the current prototype implementation of the symbolic execution engine currently only explores executions of a single thread. The Poppler case study program is single-threaded.

D. Transformation-level Path Explosion

Our approach is based on selecting a path (sequence) of transformations by which data flows from an attacker-controlled input to a vulnerability, and then searching for an input that travels that path. Because transformations represent a higher level of abstraction than program control flow, there will generally be many fewer transformation-level paths than control-flow paths that reach a potential vulnerability; this is what we observed in the case study. In general, if there are many transformation-level paths, our approach might need to be repeated to search through them. However, we believe that in many cases, a large proportion of the transformation-level paths can be used to produce an exploit. For instance, in Poppler there are many possible sequences of stream filters, but any filter could be applied to the Type 1 font exploit.

E. Exploit Weaponization

In this paper we focus on proof-of-concept (POC) exploit generation, the process of creating a program input that proves that a vulnerability is a real danger. Further steps are required to transform a POC exploit into one that could be used in a real attack, including adding an attack payload (i.e., shellcode), and hardening against defensive techniques. Because of the flexibility of symbolic execution, many existing automated techniques for these *weaponization* tasks could be incorporated naturally with our tool.

IX. Related Work

Here we mention some other recent research projects that are similar in goals, approach, or both, to our project.

Perhaps the most similar end-to-end approach is the decomposition and restitching of Caballero et al. [19]. They also tackle the problem of vulnerability conditions which are difficult to trigger directly because of other transformations the input undergoes, in their case studies decryption. Though they also used symbolic exploration to find vulnerabilities, they used a different technique to generate preimages that was based on searching for an inverse function in the same binary. The decomposition and restitching technique can also recompute checksums, which is a key capability of TaintScope [20]. TaintScope uses taint-directed fuzzing to search for vulnerabilities, and a checksum can typically be recomputed using simple concrete execution. However TaintScope uses symbolic execution, including lookup tables identified by IDAPro, to find preimages for transformations of the checksum value in a file, such as endian conversions or decimal/binary translation. Optimizations such as the others we describe in Section V-A would not be necessary here because checksum values are usually short.

Another project that shares our end-to-end goal of automatically creating exploits is the AEG system [18]. AEG also uses symbolic execution, though it takes a mixed source-code and binary approach. AEG automates a larger part of the exploit generation process, including searching for a vulnerability and generating some common kinds of jumps to shellcode. A successor system MAYHEM [21] removes the requirement for source code and introduces several optimizations. Notably MAYHEM's index-based memory model provides the same benefits as the treatment of accesses we describe in Section V-D (and was developed concurrently). However, these projects do not describe any vulnerabilities as involving transformation of the input prior to the vulnerable code, which is the key challenge we address. Beyond finding a single exploit, symbolic exploration can also be applied to the problem of characterizing all the possible exploits of a vulnerability, which is valuable for signature generation [22]; our techniques would also be valuable in this context.

The kinds of program information contained in the HI-CFG have been available separately using existing techniques; the focus of our contribution is on showing the extra value that comes from combining them in a single representation. For instance, having both information-flow and producer-consumer edges allows our approach to characterize a transformation in terms of both the data structures it operates on and the code that implements it. The program dependence graph (PDG) [3], [4] also has edges representing both control and data flow, but it is unsuitable for our application as it has no nodes representing data structures.

Our problem of computing preimages for transformations is similar to the "gadget inversion" performed by the Inspector Gadget system [23], which also applies to functionality automatically discovered within a binary. Inspector Gadget's search for inverses uses only concrete executions, but it keeps track of which output bytes depend on which input bytes. Symbolic execution can be seen as a generalization in that symbolic expressions indicate not just which input values an output value depends on, but the functional form of that dependence. This often allows symbolic execution to compute a preimage using many fewer executions.

Our technique is based on the intuition of searching backwards through the program execution to see if a vulnerability can be triggered by the input. A similar intuition has been applied to the control flow of a program (as opposed to information flow as we consider); examples include the static analysis tool ARCHER [24] and the call-chain-backward symbolic execution approach of Ma et al. [25].

Our techniques for determining which memory access constitute a buffer are most similar to the array detection algorithms of Howard [26], [27], a tool which infers data-structure definitions from binary executions. Our algorithms are somewhat simpler because we do not currently attempt, for instance, to detect multidimensional arrays. Other systems that perform type inference from binaries include REWARDS [28] which has been used to guide a search for vulnerabilities, and TIE [29] which can be either a static or dynamic analysis. Compared to these systems our HI-CFG also contains information about code and the relationships between code and data, which are needed for our application. Similar algorithms have also been used for inferring the structure of network protocols [30].

X. Conclusion

We have proposed a technique for automatically generating proof-of-concept exploits for vulnerabilities in binary x86 executables that perform input transformations. We introduce a new data structure, the Hybrid Information- and Control-Flow Graph (HI-CFG), and give algorithms for constructing a HI-CFG from traces. The HI-CFG captures the structure of buffers and transformations that a program uses for processing its input. This structure lets us perform the search for an exploit input more efficiently by applying symbolic exploration to smaller vulnerable functions and to find preimages for individual transformations. We show the feasibility and applicability of our approach in two case studies. The first study uses the Poppler PDF parsing library; our tool creates a POC exploit involving both a complex vulnerability and a decompression transformation, among other transformations. Constructing exploits that involve such complex transformation sequences would also be useful for attackers attempting to bypass signature-based filters. The second study automatically generates a POC exploit for an XML parsing error in the support for opening `.docx` XML documents in AbiWord.

Acknowledgments

We would like to thank Lenx Tao Wei for his comments and work on the HI-CFG paper and implementation. This work was supported in part by the DARPA award HR0011-12-2-005 and the AFOSR MURI award FA9550-09-1-0539.

References

[1] H. Shacham, "The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86)," in *CCS'07.*

[2] E. J. Schwartz, T. Avgerinos, and D. Brumley, "Q: Exploit hardening made easy," in *USENIX Security'11.*

[3] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *TOPLOS'87*, vol. 9, no. 3.

[4] S. Horwitz, T. W. Reps, and D. Binkley, "Interprocedural slicing using dependence graphs," *TOPLAS'90*, vol. 12, no. 1.

[5] J. Caballero, N. M. Johnson, S. McCamant, and D. Song, "Binary code extraction and interface identification for security applications," in *NDSS'10.*

[6] D. Brumley and D. Song, "Privtrans: automatically partitioning programs for privilege separation," in *USENIX Security'04.*

[7] "BitBlaze: Binary analysis for computer security," http://bitblaze.cs.berkeley.edu/.

[8] Intel, "Pin website," *http://www.pintool.org/*, Nov. 2012.

[9] M. D. Bond and K. S. McKinley, "Probabilistic calling context," in *OOPLSA'07.*

[10] V. Ganesh and D. L. Dill, "A decision procedure for bit-vectors and arrays," in *CAV'07.*

[11] E. J. Schwartz, T. Avgerinos, and D. Brumley, "All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask)," in *IEEE S&P'10.*

[12] C. Cadar, D. Dunbar, and D. Engler, "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *OSDI'08.*

[13] V. Chipounov, V. Kuznetsov, and G. Candea, "S2E: a platform for in-vivo multi-path analysis of software systems," in *ASPLOS'11.*

[14] Reference omitted for anonymous submission.

[15] MITRE, "CVE-2010-3704: Memory corruption in FoFiType1::parse," http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2010-3704, Oct. 2010.

[16] P. Deutsch, "DEFLATE compressed data format specification," IETF RFC 1951, May 1996.

[17] D. Babić, L. Martignoni, S. McCamant, and D. Song, "Statically-directed dynamic automated test generation," in *ISSTA'11.*

[18] T. Avgerinos, S. K. Cha, B. L. T. Hao, and D. Brumley, "AEG: Automatic exploit generation," in *NDSS'11.*

[19] J. Caballero, P. Poosankam, S. McCamant, D. Babic, and D. Song, "Input generation via decomposition and re-stitching: Finding bugs in malware," in *CCS'10.*

[20] T. Wang, T. Wei, G. Gu, and W. Zou, "TaintScope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection," in *IEEE S&P'10.*

[21] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley, "Unleashing mayhem on binary code," in *IEEE S&P'12*.

[22] J. Caballero, Z. Liang, P. Poosankam, and D. Song, "Towards generating high coverage vulnerability-based signatures with protocol-level constraint-guided exploration," in *RAID'09*.

[23] C. Kolbitsch, T. Holz, C. Kruegel, and E. Kirda, "Inspector Gadget: Automated extraction of proprietary gadgets from malware binaries," in *IEEE S&P'10*.

[24] Y. Xie, A. Chou, and D. R. Engler, "ARCHER: using symbolic, path-sensitive analysis to detect memory access errors," in *ESEC/FSE'03*.

[25] K.-K. Ma, Y. P. Khoo, J. S. Foster, and M. Hicks, "Directed symbolic execution," in *SAS'11*.

[26] A. Slowinska, T. Stancescu, and H. Bos, "Howard: a dynamic excavator for reverse engineering data structures," in *NDSS'11*.

[27] ——, "Body armor for binaries: preventing buffer overflows without recompilation."

[28] Z. Lin, X. Zhang, and D. Xu, "Automatic reverse engineering of data structures from binary execution," in *NDSS'10*.

[29] J. Lee, T. Avgerinos, and D. Brumley, "TIE: Principled reverse engineering of types in binary programs," in *NDSS'10*.

[30] J. Caballero, H. Yin, Z. Liang, and D. Song, "Polyglot: Automatic extraction of protocol message format using dynamic binary analysis," in *CCS'07*.