HI-CFG: Construction by Binary Analysis, and Application to Attack Polymorphism

Dan Caselden[1], Alex Bazhanyuk[2], Mathias Payer[3], Stephen McCamant[4], and Dawn Song[3]

[1] FireEye, Inc.
[2] Intel Corporation
[3] University of California, Berkeley**
[4] University of Minnesota

Abstract. Security analysis often requires understanding both the control and data-flow structure of a binary. We introduce a new program representation, a hybrid information- and control-flow graph (HI-CFG), and give algorithms to infer it from an instruction-level trace. As an application, we consider the task of generalizing an attack against a program whose inputs undergo complex transformations before reaching a vulnerability. We apply the HI-CFG to find the parts of the program that implement each transformation, and then generate new attack inputs under a user-specified combination of transformations. Structural knowledge allows our approach to scale to applications that are infeasible with monolithic symbolic execution. Such attack polymorphism shows the insufficiency of any filter that does not support all the same transformations as the vulnerable application. In case studies, we show this attack capability against a PDF viewer and a word processor.

1 Introduction

In security analysis it is often necessary to understand both the information-flow and control-flow structure of a large code base. Disassemblers concentrate on recovering control-flow structure, and some research systems [18,26,17] reverse engineer data structures. But there is insufficient automated support for understanding the flow of information between data structures, and the relationship between data structures and code. We propose new techniques that combine information-flow analysis with control-flow graph recovery to scale precise binary analysis to large software systems, and apply them to generating polymorphic attacks against programs that support complex input transformations.

Applications are getting larger and more complex due to increasing functionality, a more sophisticated software stack, and new abstractions and concepts that simplify development. These applications are hard to debug and vulnerabilities are becoming more and more complex, e.g., a vulnerable program location might only be reached after a specific input is passed through several buffers and

** The authors were all at UC Berkeley while performing the primary research.

```
1  // ISO-8859-1 to UTF-8 conversion
2  void trafo(char *src,
3             char *dst, int len) {
4    while (len-- > 0) {
5      if (*src < 0x80) {
6        *dst++ = *src++;
7      } else {
8        *dst++ = 0xc0 | \
9            (*src & 0xc0) >> 6;
10       *dst++ = 0x80 | \
11           (*src++ & 0x3f);
12     }
13   }
14 }
15 ...
16 trafo(buf0, buf1, 256);
```

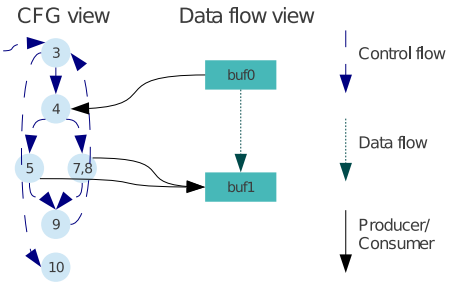

Fig. 1: This example shows both (a) simple transformation and (b) the corresponding HI-CFG.

functions whereas the data can be modified by each function. Symbolic execution is a great tool to analyze security properties of an application given a potentially vulnerable program location. Unfortunately, symbolic execution does not scale well to large contexts and long-running programs with multiple input transformations, due to the explosion of the number of possible paths that have to be evaluated and the state that has to be tracked. A simple alternative to symbolic execution is (concrete) fuzzing or fuzz testing. Fuzz testing uses templates to probabilistically generate input data that tries to trigger a program crash. A security analyst then analyzes the crash logs to locate vulnerabilities. Due to the probabilistic input generation fuzz testing is unlikely to reach a vulnerability that is guarded by complex, low-probability conditions.

This paper introduces a new program representation, a Hybrid Information- and Control-Flow Graph (HI-CFG), that captures both the information-flow graph and the control-flow graph of a program. The HI-CFG shows the data structures within a program as well as the code that generates and uses these data structures, inferring an explicit connection of producer and consumer edges between data-flow nodes and blocks in the control-flow graph. Figure 1(a) shows a simple example of a transformation and the corresponding HI-CFG graph. The transformation that copies data from buffer `buf0` to `buf1`. Figure 1(b) 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.

Using the information in the HI-CFG about individual data structures (i.e., buffers) and transformations enables an iterative, step-by-step analysis of these buffer transformations. Instead of using monolithic symbolic execution that reverses all transformations in a single (but potentially exponentially large) step, iterative symbolic execution starts from a potentially vulnerable program location and reverses each transformation individually. Figure 2 shows a vulnerability hidden behind several transformations that can be reversed using iterative symbolic execution.

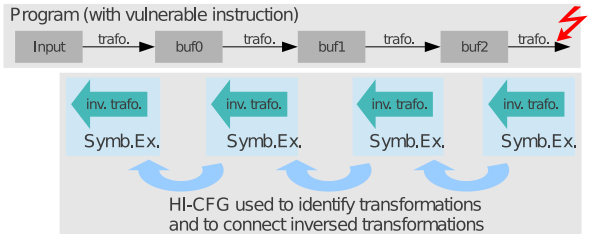

Fig. 2: Iteratively reversing a set of data transformations based on information from the HI-CFG.

One possible way to build a HI-CFG is using source-based program analysis. Unfortunately, in the context of security analysis, the source-code of a program is usually not available and the program itself might be stripped. This paper presents an algorithm to build a HI-CFG for a given binary-only program based on the analysis of an execution trace for a benign input that executes the buffer transformations but does not necessarily trigger the vulnerability. Prerequisites for the algorithm are only the (stripped) binary and a benign input that executes the buffer transformations.

Another advantage of the HI-CFG from an attacker's perspective is that given one vulnerability the symbolic execution engine can be used to generate many different exploit paths, leveraging different encodings or different transformations. Often transformations are many-to-one (e.g., many different deflate compressed streams decode to the same original data) and the symbolic execution engine can be used to produce different encodings for a specific target string. Also, many file formats allow a specific program location to be reached by different chains of transformations. With file formats that allow recursive objects, an attacker can choose from an infinite amount of transformations. Such attacks can only be detected if the analysis tool has deep knowledge of the file format and implements all transformations as well.

We evaluate the feasibility of HI-CFG construction using only a stripped binary for two case studies: a PDF viewer, and a word processor. For both programs, we describe the construction of the HI-CFG as well as how symbolic execution can be used to generate different attacks by inverting transformations along the HI-CFG buffer chains.

The contributions of this paper are:

1. we introduce a new program representation, a *Hybrid Information- and Control-Flow Graph (HI-CFG)*, which combines control-flow and data-flow information by inferring producer/consumer edges;
2. we give algorithms for building a HI-CFG given only a stripped binary program and a benign input to that program;
3. we evaluate the security capabilities of the HI-CFG using two case studies for large, real-world programs: Poppler and AbiWord.

Our symbolic execution approach for attack generation is not a contribution here; it is described in more detail in a technical report [21]. Further information about the project is available on the BitBlaze web site [12].

2 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.

We start by describing the kinds of nodes and edges found in a HI-CFG (Section 2.1). Then we mention potential variations of the concept and applications for which they would be suitable (Section 2.2).

2.1 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 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 graph. However, the HI-CFG is more powerful than a simple combination of a

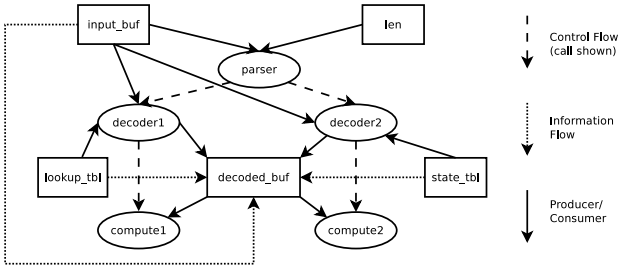

Fig. 3: A detailed example of a coarse-grained HI-CFG for a program which parses two kinds of commands from its input, decodes those commands using lookup tables, and then performs an appropriate computation for each command.

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.

2.2 Generality and Uses

We can create a HI-CFG with differing levels of granularity for code and data. A fine-grained code representation has one code block per basic block, while a coarse-grained representation has one code block per function. Analogously, a fine-grained data representation has a data structure node for each atomic value (like an integer), while a coarse-grained data representation has 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.

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. Data structures connected to multiple areas represent the interfaces of those components.

Information-flow Isolation. A different kind of decomposition would be valuable for 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 part of an application needs to access sensitive information directly. Just as automatic privilege separation [4] 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.

Attack Generation. For this paper, our application of the HI-CFG is to find the structure of a program's buffer usage to facilitate efficient attack 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 3, 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 attack, on an analyst-chosen benign execution. If desired the analyst can repeat the search with a benign input that exercises different parts of the program functionality.

3 Dynamic HI-CFG Construction

In this section, we describe our approach to HI-CFG construction: first some infrastructure details, then techniques for 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.

3.1 Infrastructure

To construct a HI-CFG via dynamic analysis, we take a trace-based approach. We use the BitBlaze Tracecap tool to record instruction traces. Tracecap also records statistics about loaded executables and libraries, 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 [15] library (for instruction decoding). 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. The implementation of the trace collection and trace analysis focuses on x86 while the techniques for HI-CFG construction apply to general architectures.

3.2 Control Flow

The HI-CFG construction module primarily identifies functions by observing `call` and `ret` in the instruction trace. At a `call` instruction, the module updates the call stack for the current thread and creates a control-flow edge from the caller to the callee. (This includes indirect `call` instructions such as those used for C++ virtual methods.) At a `ret`, 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 previously been `call` targets. A limitation of this approach is that tail-called functions will never be recognized if not normally 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.

3.3 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 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

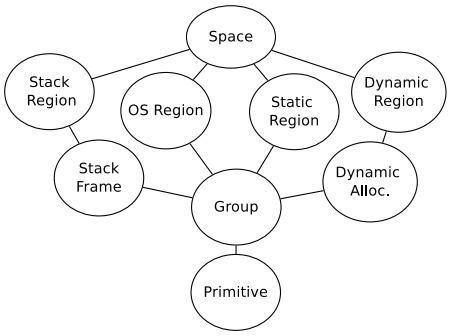

Fig. 4: The hierarchy of types in the model of memory used in our HI-CFG construction algorithm.

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. Additional constraints prohibit stack frames and dynamic allocations from appearing in the memory model without their respective regions.

3.4 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. Specifically, our system numbers the accesses sequentially and then sorts them by their address. A long enough run of adjacent accesses (currently 6) form a group if for each pair of adjacent accesses, the distances between them, both in the sequential order and in address, match. A pseudocode description of this algorithm, simplified to omit the treatment of nested functions and some optimizations, is in Figure 5. An example of the algorithm applied to `strcpy` can be found in a companion technical report [8]. 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. If found, the system will submit the candidate buffer for further processing described next.

```
const MIN_SIZE = 6
for insn in trace
  match insn.type of
    case CALL: opcount := 0; accesses := [ ]
    case LOAD(addr) or STORE(addr):
      append(accesses, (addr, opcount++))
    case RETURN:
      a := sort(accesses, by(addr))
      for indexes i in a
        group := [ a[i] ]
        old_stride := None
        for indexes j=i, j+1 in a
          new_stride := (a[j+1].addr − a[j].addr, a[j+1].opcount − a[j].opcount)
          if (old_stride == None) old_stride := new_stride
          if (old_stride ≠ new_stride) break
          append(group, a[j+1])
        if (group.length ≥ MIN_SIZE) make_group(group)
```

Fig. 5: Pseudocode for identifying linear access patterns in a trace.

Once the two systems have submitted their candidate buffers, the HI-CFG module combines the sets of discovered buffers (keeping all non-overlapping buffers, and preferring larger buffers in the case of overlap) 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. Our system merges a subsequent buffer with an existing one if the starting or ending addressing of the new buffer matches either the starting or ending address of the old one, or if either buffer is completely contained within the other.

3.5 Information Flow

To trace the information flow between buffers, our system primarily uses a specialized form of dynamic taint analysis [23,24]. 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, or used in arithmetic or bitwise operations. 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. Therefore, we supplement it with an upper-bound technique that constructs a low-confidence information-flow edge whenever the temporal sequence of buffers consumed and produced by a function would allow an information flow. (In other words, if a function first reads from buffer A, and then later writes to buffer B, we create a low-confidence information flow edge from A to B.)

3.6 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 choice 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 and the same key. In the current implementation we store an MD5 hash of the key material to save space. The key includes an identifier for the type of an object, and by default it also contains the object's offset within its parent.

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 [3]. Stack frames use the address of the function. As a result, our system is able to identify two local variables or dynamic allocations as the same across multiple calls to a function and in the presence of custom memory allocation wrappers.

We use a disjoint-set union-find data structure [13] 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.

4 Application: Attack Polymorphism

As our primary example of a security application of a HI-CFG, we describe how to use the transformation structure represented in the HI-CFG to efficiently produce new attacks that differ in the transformations applied to the input before reaching a vulnerability. We first describe the technique and how it uses the HI-CFG, then describe experiments applying the technique to two vulnerable document-processing applications.

4.1 Transformation-Aware Attack Generation with a HI-CFG

In a large application, an input value will typically undergo a number of transformations before being used in a vulnerable function. Moreover, the sequence of transformations that apply may vary depending on the input structure. For instance portions of a document might appear in one of several encoding formats, or they might be compressed. This flexibility is potentially powerful for an attacker, because it allows for polymorphism: the same underlying attack can be carried out using a wide variety of input files which look superficially dissimilar.

We show that using the transformation structure available in the HI-CFG, along with symbolic execution, an attacker can easily generate transformed attack inputs, without a need to understand the transformations. We treat the generation of transformed inputs as a search problem, and we use the structure of transformations to guide the search. Symbolic execution does not scale to generate complete inputs to a large program. But using the transformation structure, we can apply symbolic execution to search for a pre-image of a single transformation at a time.

Specifically, our approach generates a HI-CFG from an execution of the vulnerable program on a benign input which does not contain an attack, but does exercise the desired transformations. We also presume that the attacker has enough knowledge to trigger the attack in the vulnerable function (perhaps also by symbolic execution); in general this is not enough to directly give a program input that triggers the vulnerability. Our system uses the transformation structure from the HI-CFG to determine the relevant transformations performed on

	Test 1	Test 2	Test 3	Test 4	Test 5
Hex encoded	10	16	55	125	250
RLE encoded	5	8	25	60	120
Object data	12	10	29	57	114

Table 1: Set-up of the different test cases. All values are in bytes.

the program input to produce the buffer contents used by the vulnerable function. It then uses repeated searches based on symbolic execution, working backward from the vulnerable function's input buffer. For each transformation, it computes a pre-image: buffer contents for a previous buffer, which when passed through the transformation, yield the contents for the next buffer in the transformation sequence. This process is shown graphically in Figure 2.

A sequence of transformations leading to the function containing a potential vulnerability will appear in the HI-CFG as a path. 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. There may be multiple such paths, such as if there are buffers containing both primary data and meta-data. Among all the paths of the form described above, we choose the path for which the size of the smallest buffer on the path is maximized, to prefer primary data buffers.

Given the sequence of buffers, the HI-CFG also contains information about which functions implement each transformation. 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.

4.2 Performance Comparison between Iterative and Monolithic Symbolic Execution

This section empirically evaluates the proposition that iteratively reversing individual transformation is faster than reversing all transformations in one single (but more complex) step.

Our test program sets up a chain of two transformations. The input is first hex decoded (pairs of ASCII characters in the ranges 0–9, a–f, or A–F map into data bytes, skipping whitespace). The data bytes are then decompressed according to a byte-level run length encoding (RLE), in which compressed bytes indicate either a repeat count for a single byte, or a run of bytes to be copied verbatim. Both encoding schemes are supported for objects in PDF files: in sequence they encode data that is compressed but still printable.

For our performance evaluation we use three different configurations of the same application with different input data. See Table 1 for the different test

	Test 1 [s]	Test 2 [s]	Test 3 [s]	Test 4 [s]	Test 5 [s]
Iterative SE	20	183	77	816	37635
Monolithic SE	599	14042	35972	Timeout	Timeout

Table 2: Scalability of iterative symbolic execution compared to monolithic symbolic execution. All numbers are in seconds, the timeout was set to 12 hours.

configurations of the data that is used for the two transformations. We then evaluate both iterative and monolithic symbolic execution. Monolithic symbolic execution uses the object data as its target and directly recovers the hex encoded input data. Iterative symbolic execution leverages the HI-CFG representation to split up the large transformation into two transformations and recovers the RLE encoded data first and uses the result from the first step as input for the second step where the RLE encoded data is reversed to hex encoded data.

The experiments use our binary symbolic execution tool FuzzBALL [2,20], which builds in turn on the Vine library from the BitBlaze framework [27]. To further improve its performance on generating transformation pre-images, FuzzBALL includes support for pruning unproductive paths, prioritizing paths by the prefix length they produce, and handling loads and stores to tables with single large formulas. These are described in detail in a technical report [21], and the implementation is available from the BitBlaze web site. To isolate the benefit of the HI-CFG, we enable these other optimizations for monolithic symbolic execution as well.

Table 2 shows the different performance for iterative and monolithic symbolic execution. Even for very short input sequences with only few bytes as object data iterative symbolic execution clearly outperforms monolithic symbolic execution by 30x (for Test 1). For larger test cases iterative symbolic execution outperforms monolithic symbolic execution by up to 78x (Test 2) or 467x (Test 3).

4.3 Case Studies

As case studies, we apply our attack polymorphism to two vulnerable document-processing systems: the PDF parsing library Poppler and the word processor AbiWord. 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.

Poppler Poppler is a PDF processing library used in applications such as Evince. The vulnerability for which we generate attacks is cataloged as CVE-2010-3704 [22]. The vulnerability is an integer overflow in a Type 1 font character index, which can trigger an arbitrary memory write. 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 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.

We can also apply symbolic execution to create the malicious font itself; details are in a previous technical report [8]. We used a separate benign input and generated a separate HI-CFG for each sequence of transformations. For space reasons we give a detailed description of the first; the others were similar.

The most common PDF compression format is `FlateDecode`, using the Deflate algorithm of RFC 1951 [10]. As a benign input, we use a PDF file generated by `pdftex` applied to a small TEX file, which contains a FlateDecode-compressed font. The execution trace from the benign execution contains 13,560,478 instructions, and constructing the HI-CFG took about 1.2 hours (4217 s) on a Xeon X5670. The HI-CFG contains 1283 functions and 1590 groups.

An excerpt of the relevant portion of the HI-CFG generated by our tool is shown in Figure 6. Input passes through a sequence of four buffers before the vulnerable code is triggered, so given contents for the final buffer which trigger a vulnerability in the font parser, our system compute three levels of preimages. However, two of the transformations are direct copies for which preimage computation is trivial. Between the second and third buffers our system computes a preimage under the FlateDecode transformation: a compressed font that decompresses to the attack font. One average this requires searching through 111 execution paths, and takes a little less than two hours (6598.69 s over ten runs dropping the fastest and slowest, on an Intel Core 2 Duo E8400).

Another commonly-used transformation of streams in PDF files is RC4 encryption. It is relatively easy for our symbolic execution system to re-encrypt modified data by constructing pre-images because RC4 is a stream cipher, and the key is fixed. We applied our technique to a version of the previously described sample document with RC4 and an owner password. There is one symbolic path, and the running time is 20 seconds, mostly devoted to program startup.

Two further transformations supported by Poppler include run-length encoding and a hexadecimal encoding of binary data, as described in Section 4.2. 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. As seen with the implementation in Section 4.2, these transformations are relatively easy to invert; the preimage computation requires 143 seconds and 315 symbolic paths.

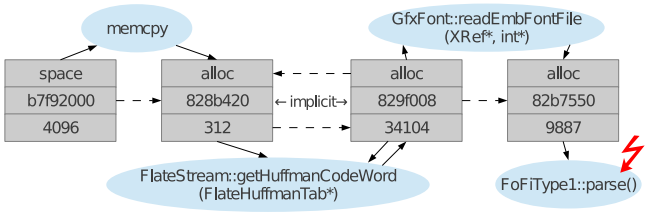

Fig. 6: An excerpt of the HI-CFG for our Poppler case study showing the buffer sequence. The input travels from left to right and FoFiType1::parse contains the vulnerability.

AbiWord AbiWord is a word-processing application that supports a number of file formats. In particular we examined its processing of documents in Office Open XML format (used with the extension `.docx`) from 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; we have not determined whether this bug is exploitable. The execution trace collected from the benign execution contains 69,503,117 instructions, and constructing the HI-CFG took about 5.8 hours (20910 s). The generated HI-CFG contains 5379 functions and 5838 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 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. An XML document triggering the crash could be found using a schema-aware random testing tool, or the details of the attack can also be completed using symbolic execution of the parser [8].

Given the crash-inducing XML text, our tool finishes the task of producing an attack `.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. On average (across 10 runs dropping the fastest and slowest), the search requires 237 seconds and 92 symbolic paths.

4.4 Discussion

Next we discuss in more detail some of the limitations and implications of the attack polymorphism capability.

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 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. 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.

Implications for Attack Filters. Our results show that it is easy for an attacker to create variants of an attack that are camouflaged using transformations supported by an input format, such as the various filters supported in PDF documents. The consequence for the designers of defenses such as network

sensors and anti-virus systems is that in order to recognize all the variants of an attack, these systems would have to duplicate all of the transformations implemented in the system they protect. For instance to recognize all possible variants of an attack PDF, a defense system would need to include decoders for all the stream formats supported by Adobe Reader.

5 Related Work

Our techniques for determining which memory accesses constitute a buffer are most similar to the array detection algorithms of Howard [26,25], 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 [18] which has been used to guide a search for vulnerabilities, and TIE [17] which can be either a static or dynamic analysis. Similar algorithms have also been used for inferring the structure of network protocols [7]. By contrast, our HI-CFG also contains information about code and the relationships between code and data, which are needed for our application.

Perhaps the most similar end-to-end approach to attack generation is the decomposition and restitching of Caballero et al. [6]. They also tackle the problem of vulnerability conditions which are difficult to trigger because of other transformations the input undergoes, in their case studies decryption. Though they use symbolic exploration to find vulnerabilities, they use a different technique, based on searching for an inverse function in the same binary, to generate preimages. The decomposition and restitching technique can also recompute checksums, which is a key capability of TaintScope [28]. 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 simple transformations of the checksum value in a file, such as endian conversions or decimal/binary translation.

The AEG [1] and MAYHEM [9] systems also generate attack inputs using symbolic execution. AEG automates some additional aspects of exploit generation not covered in this paper, such as generating some common kinds of jumps to shellcode. 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.

The kinds of program information contained in the HI-CFG are available separately using existing techniques; the focus of our contribution is 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) [11,14] 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 Inspector Gadget [16], 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 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 [29] and the call-chain-backward symbolic execution approach of Ma et al. [19].

6 Conclusion

In this paper we introduce a new data structure, the Hybrid Information- and Control-Flow Graph (HI-CFG), and give algorithms for constructing a HI-CFG from binary-level traces. The HI-CFG captures the structure of buffers and transformations that a program uses for processing its input. This structure lets us generate transformed attack inputs efficiently, because understanding the structure of transformations allows our system to find preimages for them one-by-one. We show the feasibility and applicability of our approach in two case studies of the Poppler PDF library and the AbiWord word processor. This demonstrated ease of constructing attacks using complex transformation sequences implies that the problem of filtering such attacks is very difficult.

Acknowledgments. We thank László Szekeres and Lenx Tao Wei for suggestions and help related to the experiments and previous papers. This work was supported by NSF awards CCF-0424422, 0842695, and 0831501; MURI awards N000140911081 (ONR) and FA9550-09-1-0539 (AFOSR), and DARPA award HR0011-12-2-005. However the findings and conclusions are those of the authors and do not necessarily reflect the views of the NSF or other supporters.

References

1. Avgerinos, T., Cha, S.K., Hao, B.L.T., Brumley, D.: AEG: Automatic exploit generation. In: NDSS'11
2. Babić, D., Martignoni, L., McCamant, S., Song, D.: Statically-directed dynamic automated test generation. In: ISSTA'11
3. Bond, M.D., McKinley, K.S.: Probabilistic calling context. In: OOPLSA'07
4. Brumley, D., Song, D.: Privtrans: automatically partitioning programs for privilege separation. In: USENIX Security'04
5. Caballero, J., Johnson, N.M., McCamant, S., Song, D.: Binary code extraction and interface identification for security applications. In: NDSS'10
6. Caballero, J., Poosankam, P., McCamant, S., Babic, D., Song, D.: Input generation via decomposition and re-stitching: Finding bugs in malware. In: CCS'10

7. Caballero, J., Yin, H., Liang, Z., Song, D.: Polyglot: Automatic extraction of protocol message format using dynamic binary analysis. In: CCS'07
8. Caselden, D., Bazhanyuk, A., Payer, M., Szekeres, L., McCamant, S., Song, D.: Transformation-aware exploit generation using a HI-CFG. Tech. Rep. UCB/EECS-2013-85, University of California, Berkeley (May 2013)
9. Cha, S.K., Avgerinos, T., Rebert, A., Brumley, D.: Unleashing MAYHEM on binary code. In: IEEE S&P'12
10. Deutsch, P.: DEFLATE compressed data format specification. IETF RFC 1951 (May 1996)
11. Ferrante, J., Ottenstein, K.J., Warren, J.D.: The program dependence graph and its use in optimization. TOPLAS'87 9(3)
12. HI-CFG project information page, `http://bitblaze.cs.berkeley.edu/hicfg/`
13. Hopcroft, J.E., Ullman, J.D.: Set merging algorithms. SIAM J. Comput. '73 2(4)
14. Horwitz, S., Reps, T.W., Binkley, D.: Interprocedural slicing using dependence graphs. TOPLAS'90 12(1)
15. Intel: Pin website. `http://www.pintool.org/` (Nov 2012)
16. Kolbitsch, C., Holz, T., Kruegel, C., Kirda, E.: Inspector Gadget: Automated extraction of proprietary gadgets from malware binaries. In: IEEE S&P'10
17. Lee, J., Avgerinos, T., Brumley, D.: TIE: Principled reverse engineering of types in binary programs. In: NDSS'11
18. Lin, Z., Zhang, X., Xu, D.: Automatic reverse engineering of data structures from binary execution. In: NDSS'10
19. Ma, K.K., Khoo, Y.P., Foster, J.S., Hicks, M.: Directed symbolic execution. In: SAS'11
20. Martignoni, L., McCamant, S., Poosankam, P., Song, D., Maniatis, P.: Path-exploration lifting: Hi-fi tests for lo-fi emulators. In: ASPLOS'12
21. McCamant, S., Payer, M., Caselden, D., Bazhanyuk, A., Song, D.: Transformation-aware symbolic execution for system test generation. Tech. Rep. UCB/EECS-2013-125, University of California, Berkeley (Jun 2013)
22. MITRE: CVE-2010-3704: Memory corruption in FoFiType1::parse. `http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2010-3704` (Oct 2010)
23. Newsome, J., Song, D.: Dynamic taint analysis: Automatic detection, analysis, and signature generation of exploit attacks on commodity software. In: NDSS'05
24. Schwartz, E.J., Avgerinos, T., Brumley, D.: 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
25. Slowinska, A., Stancescu, T., Bos, H.: Body armor for binaries: preventing buffer overflows without recompilation. In: USENIX ATC'12
26. Slowinska, A., Stancescu, T., Bos, H.: Howard: a dynamic excavator for reverse engineering data structures. In: NDSS'11
27. Song, D., Brumley, D., Yin, H., Caballero, J., Jager, I., Kang, M.G., Liang, Z., Newsome, J., Poosankam, P., Saxena, P.: BitBlaze: A new approach to computer security via binary analysis. In: ICISS'08, keynote invited paper
28. Wang, T., Wei, T., Gu, G., Zou, W.: TaintScope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection. In: IEEE S&P'10
29. Xie, Y., Chou, A., Engler, D.R.: ARCHER: using symbolic, path-sensitive analysis to detect memory access errors. In: ESEC/FSE'03