# Runtime Execution Tracing and Alignment with PANDA

by

Leah Goggin

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Masters of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2019

Author . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
Department of Electrical Engineering and Computer Science
August 23, 2019

Certified by . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
Tim Leek
Technical Staff, MIT Lincoln Laboratory
Thesis Supervisor

Accepted by . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
Katrina LaCurts
Chair, Master of Engineering Thesis Committee

# Runtime Execution Tracing and Alignment with PANDA

by

Leah Goggin

Submitted to the Department of Electrical Engineering and Computer Science
on August 23, 2019, in partial fulfillment of the
requirements for the degree of
Masters of Engineering in Electrical Engineering and Computer Science

## Abstract

When a reverse engineer executes a binary multiple times and observes different behaviors, it is often not obvious what caused the difference, since the two executions may diverge much earlier for unrelated reasons and the relevant input may be one among tens of millions. This thesis implements a plugin for the Platform for Architecture-Neutral Dynamic Analysis (PANDA) which tracks control flow at runtime at the basic block level, recording a hash of the current location in program structure at every conditional jump. By considering only control flow history up to and including the current stack frame, multiple divergences and reconvergences can be detected. Comparing these hashes across executions allows the engineer to narrow down branches of interest and request more detailed information from the plugin on a second pass, during which the precise context of nested functions and control flow within functions is recorded at the selected moment of divergence. Existing PANDA tools may then be used to trace the variables relevant to the conditional branch back to some initial input.

Thesis Supervisor: Tim Leek
Title: Technical Staff, MIT Lincoln Laboratory

# Acknowledgments

I would like to thank Tim Leek for his time and expertise in supervising this project. Additionally, thank you to Ray, Andrew, Tiemoko, Professor Brendan Dolan-Gavitt, and the rest of the past and present PANDA team for their support throughout.

# Contents

# List of Figures

# Chapter 1

# Introduction

Typical modern software is complex enough that running it with total visibility of the instructions being executed, the program's internal state, the inputs and outputs, and so forth may not easily offer full insight into what the program is doing. Even in a case where the reverse engineer has the source code - for example, they may be looking for the root cause of some undesired behavior in their own code - it is not always obvious where amid what may be hundreds of millions of executed instructions and gigabytes of constantly-changing memory things are going wrong.

Software reverse engineering techniques are generally divided into the classes of static and dynamic analysis. Static analysis does not involve executing the code, instead focusing on other sorts of analysis. It often involves disassembling and/or decompiling the code and inspecting it for characteristics of interest, such as exploitable errors (unsafe bounds checking, etc.). Dynamic analysis does involve running the code, often with some sort of instrumentation. It offers visibility of the program's runtime state, such as call stacks and intermediate data states.

This thesis introduces a dynamic analysis tool to assist the reverse engineer in identifying instructions and data of interest. In particular, it locates the points at which two similar executions of the same code diverge, at which point the reverse engineer may use other tools to track the intermediate variable causing the divergence back to some raw input.[10][7] An example use case would be two runs of a malware binary, where in one run the malware displays some behavior such as sending a packet to a command and control server and in the other it does not, and the engineer wants to know what the malware "looked for" in order to decide what to do.

A naive solution to this problem is to simply log the program counter of every single executed instruction, then use some alignment tool such as `vimdiff` to find divergence. This technique has several problems. First, logging even a four-byte program counter for every executed instruction will generate a prohibitively large log for long executions. Second, this technique has no concept of context. That is, if the same function is called from multiple different areas of the code, it will still appear as a sequence of identical executions, making meaningful alignment difficult.

Another solution is Bin et al.'s concept of *execution indexing*, where every execution of a given instruction is associated with a unique index expressing not only its function context, but its context within the control flow inside the function, i.e. its index in any loops surrounding it and its location within any if/else blocks. Their concept of an execution index works well as a trace for tracking divergence and reconvergence. However, their solution requires static analysis on source code. Their paper is discussed in more detail in Chapter 3.

The tool presented in this thesis attempts to find a middle ground between these approaches. An index conceptually resembling, although not identical to, the one developed by Bin et al. is tracked at runtime, generating a trace that can be correlated between multiple different executions of the same code. No access to source code is necessary. Because the index is expressed as a hash recorded at every conditional control flow instruction, the size of the log is kept reasonable.

Implemented as a plugin to the PANDA reverse engineering platform, it works in two stages: first, the control flow is tracked in two or more executions of the program in which different behaviors are observed. These traces are compared and divergences are identified. Next, the locations of these divergences are passed into the plugin for a second run. This time, the plugin logs the full control flow data structure at the divergence locations. This ultimately allows the reverse engineer to see not only the precise instruction at which the executions diverged, but also the full context of that instruction.

Chapter 2 of this thesis describes the tools used, particularly QEMU and PANDA.

Chapter 3 elaborates on the concepts used from Bin et al.'s work and describes how the execution trace used in this thesis differs from and builds on it.

Chapter 4 describes practical implementation details, including how PANDA is used to instrument a running binary.

Chapter 5 describes two test cases and discusses results in terms of accuracy and usability.

Chapter 6 concludes with an assessment of future work to be done in the area.

# Chapter 2

# Background

## 2.1 QEMU

QEMU is an open-source emulator supporting both full-system and Linux user mode emulation.[4] It allows Linux, Windows, and macOS hosts to run other OSes, without requiring the guest OS to be built for the same CPU. QEMU is comprised of several subsystems, including machine descriptions and emulated and generic devices. The subsystem most relevant to this paper is the CPU emulator, which must handle code translation between different architectures and the cacheing of that code, memory management, and hardware interrupt and exception support, among other concerns.[4]

The current implementation of the code translation process first converts guest instructions into Tiny Code Generator (TCG) operations, resembling a simplified, optimized instruction set. These operations are then converted into host instructions. This simplifies the addition of new architectures, whether as guests or hosts, because it is only necessary to implement the translations from the new instruction set to/from TCG rather than implementing a translation for every combination of supported architectures.

### 2.1.1 Basic blocks

The TCG documentation defines a basic block as “a list of instructions terminated by a branch instruction”, that is, an unconditional jump, conditional jump, call, or return.[4] A basic block is assumed to start either 1. immediately after the end of the previous basic block (in terms of program counter, not execution order) or 2. at a `set_label` instruction. The idea is that once a basic block begins executing, it will run sequentially to its end, at which point QEMU determines which basic block (referred to as *translated block*, or TB) should execute next. It is possible for the execution of a basic block to be interrupted in the middle, if the block is user code and some circumstance causing a jump to kernel mode occurs. An example of this is described at the end of Section 4.1.2. Once translated, basic blocks are cached, such that ideally every basic block that executes is translated exactly once, even if it executes many times.

## 2.2 PANDA

The tool described in this thesis is implemented as a PANDA plugin. PANDA is an open-source reverse engineering platform built on QEMU and primarily developed by Lincoln Laboratory and Northeastern University. PANDA offers two major capabilities: recording and replaying executions and allowing the reverse engineer to develop custom analysis plugins. [5]

The record-and-replay capability is accomplished by storing both an initial snapshot of the state of the guest at the start of execution and a log of all nondeterministic information that crosses an “invisible line” surrounding the CPU and RAM, as in Figure 2-1. Information coming into the CPU through ports, hardware interrupts and their parameters, and direct memory accesses by peripheral devices are recorded in this log. This capability is significant because programs may behave very differently across runs if their execution is influenced by sources of nondeterminism, which often originate from input from the network or user. Giving the reverse engineer the capability to iterate on an identical run of the program allows them to more easily track

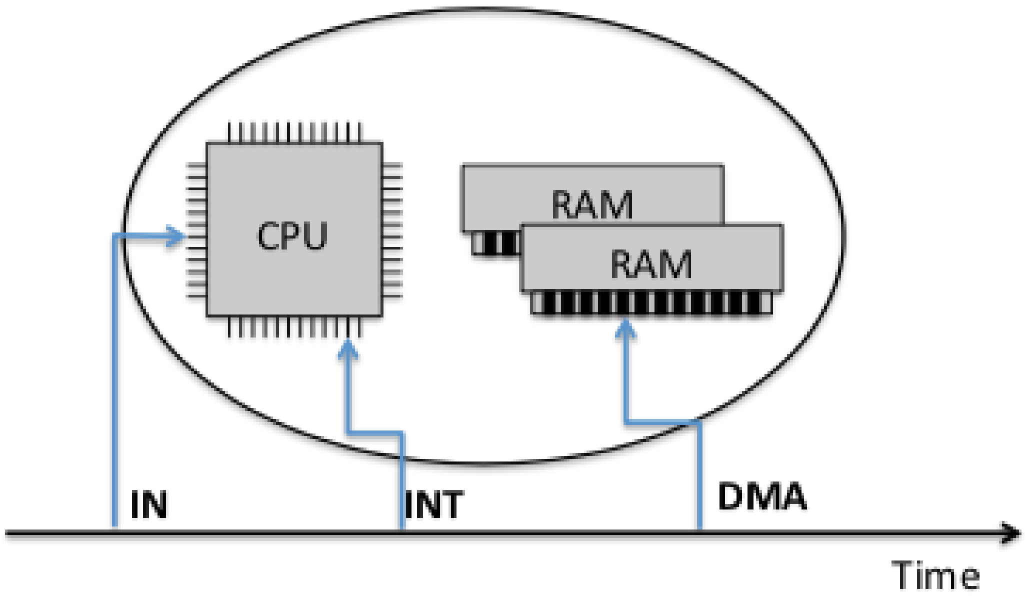


Figure 2-1: PANDA's nondeterminism boundary (Dolan-Gavitt et al.)

cause and effect and isolate variables to build up knowledge.

PANDA's plugin interface lets the reverse engineer add instrumentation at several points of interest: at the translation and execution of a particular instruction, before and/or after the translation and/or execution of every basic block, and at memory accesses, among others.[5] User-implemented callback functions have access to information such as the CPU state, the memory, and the contents of the current basic block, and may write to memory and flush the cache of translated basic blocks. Plugins are implemented in C/C++.

# Chapter 3

# Execution Indexing

An *execution index* is a particular point in a program's execution. It is more specific than a program counter, since the instruction at a given program counter may be executed many times. The precise definition of an execution index is somewhat subjective and context-dependent. Bin et al. note that "it is machine undecidable to conclude if two execution points in two respective executions correspond to each other. In practice, programmers often decide the correspondence according to their understanding of the executions. For example, if the two executions have their inputs overlapped, the sub-executions driven by the overlapping input elements should correspond."[11]

This chapter first describes Bin et al.'s approach to defining an execution index. At a high level, they fully describe a location in the program's control flow graph using a formal grammar and propose a method of deriving that grammar via static analysis. The second section describes divergences from this approach chosen to suit the use case of interest to this thesis. In particular, 1. control flow in this thesis is considered in terms of assembly rather than source code structures, 2. all analysis is done at runtime, and 3. the index must be useful and practical in a reverse engineering context.

| | | | | |
|---|---|---|---|---|
| Code | 1 $s_1$;<br>2 $s_2$;<br>3 $s_3$;<br>4 $s_4$; | 1 if (...)<br>2 $s_1$;<br>3 else<br>4 $s_2$; | 1 while (...) {<br>2 $s_1$;<br>3 }<br>4 $s_2$; | 1 void A() {<br>2 B();<br>3 }<br>4 void B() {<br>5 $s_1$;<br>6 } |
| EDL | $S \longrightarrow \widetilde{1}\,\widetilde{2}\,\widetilde{3}\,\widetilde{4}$ | $S \longrightarrow \widetilde{1}\ \mathit{R1}$<br>$\mathit{R1} \longrightarrow \widetilde{2} \mid \widetilde{4}$ | $S \longrightarrow \widetilde{1}\ \mathit{R1}\ \widetilde{4}$<br>$\mathit{R1} \longrightarrow \widetilde{2}\,\widetilde{1}\ \mathit{R1} \mid \epsilon$ | $S \longrightarrow \widetilde{2}\ \mathit{RB}$<br>$\mathit{RB} \longrightarrow \widetilde{5}$ |
| Str. | 1 2 3 4 | 1 2<br>1 4 | 1 2 1 4<br>1 2 1 2 1 4 | 2 5 |

Figure 3-1: Formal grammar rules for common source code control structures (Bin et al.)

## 3.1 Efficient Program Execution Indexing (EPEI)

In *Efficient Program Execution Indexing*, Bin et. al. propose a formal definition for an execution index: they require that for two different execution points in the same program, run on the same input, the indices must not be equal.[11] They then present an example of such an index. Their key insight is that valid paths through the control flow graph of a given piece of software can be described by a string generated from what they call an Execution Description Language (EDL), and the nested rules of this language at a given point form a suitable execution index. The grammar rules corresponding to common control flow constructs are shown in Figure 3-1.

Their paper contributes an algorithm for statically analyzing source code in order to generate the grammar rules of its particular EDL. Their EDLs use lines of source code as terminal characters and maintain a context stack during runtime to track which grammar rule should be "active" in parsing encountered code. Dominators and postdominators of function, loop, and if/else contexts are instrumented with `enter` and `exit` labels before runtime as a cue to push to and pop from this context stack. Identifying these dominators and postdominators requires static analysis of the source code prior to runtime.

At the beginning of execution, the execution index would simply be the starting

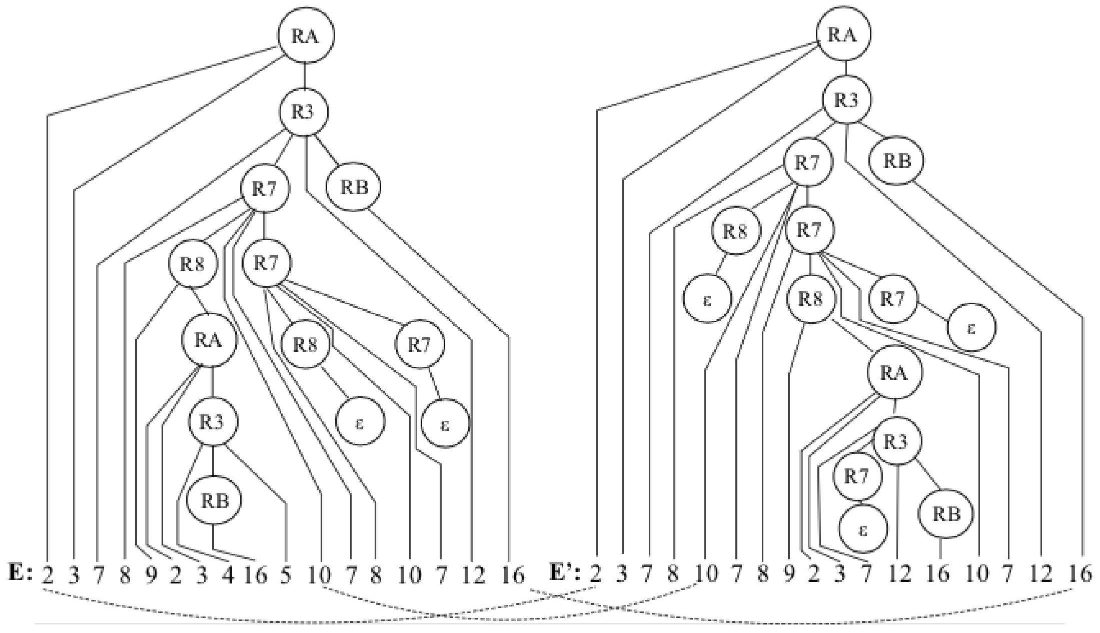

RA
R3
R7
RB
R8
ε
E: 2 3 7 8 9 2 3 4 16 5 10 7 8 10 7 12 16
E': 2 3 7 8 10 7 8 9 2 3 7 12 16 10 7 12 16

Figure 3-2: Rule derivation trees for two executions (Bin et al.)

rule. As nested contexts (of functions, loops, and if/else statements) are entered and exited, rules are pushed and popped from the stack. The contents of the stack at any given moment form the efficient program execution index (EPEI). This is equivalent to "the path from the root of the derivation tree to the leaf node representing [the point]", if the rule expansion is represented with a tree as in Figure 3-2.

The advantage of Bin et al.'s approach is that its stack-centric structure allows for reconvergence, since there is the possibility of divergent sections of execution being popped from the stack and returning it to an aligned state. The disadvantage in a reverse engineering context is that maintaining this rule stack requires knowing exactly when structures such as loops and if/else blocks are entered and exited, and Bin et al.'s preprocessing to determine this requires access to source code. The next section describes this thesis's approach to defining an index which retains the desirable qualities of the EPEI, while handling control flow within functions without requiring access to source code.

Bin et al.'s implementation also includes the concept of *anchor points.* Anchor points are user-specified locations at which the running index is reset, allowing the

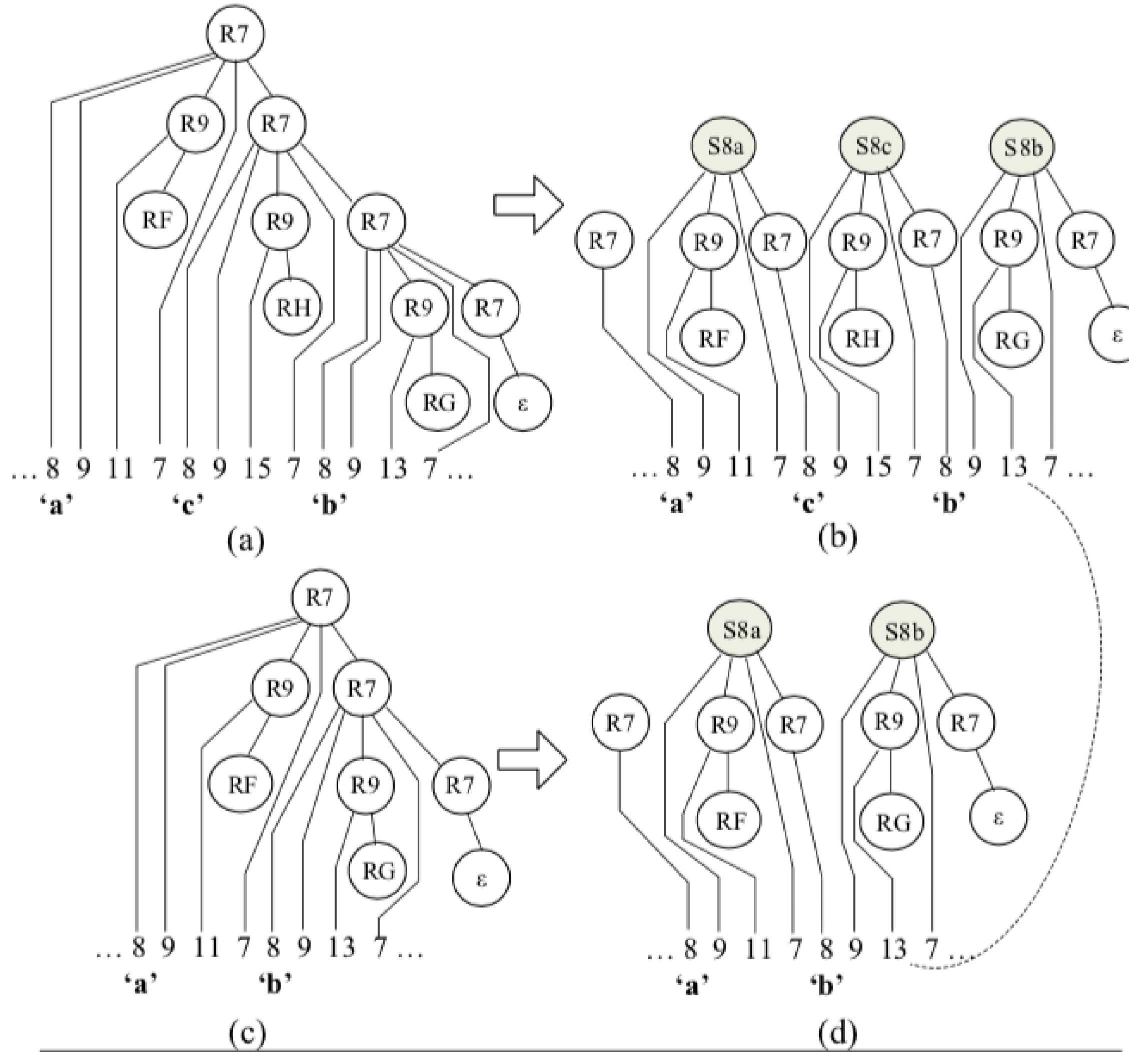


Figure 3-3: Indexing executions with input "acb" and "ab" (Bin et al.)

system to detect sequences of identical execution even if there is some divergence prior to the beginning of the aligned sequence, i.e. lower on the rule stack. In Figure 3-3, the expansion of the grammar rules into lines of source is shown for the inputs *"acb"* and *"ab"*, demonstrating how a "reset" at line 8 (the beginning of a sequence in which input letters are individually processed) allows for identical derivation trees to be generated by the identical sub-inputs. *Not* manually specifying anchor points means diverging executions can reconverge only upon exit from the context of the "lowest" divergence, which is a weakness of the stack-based indexing approach.

## 3.2 Efficient Binary Execution Indexing (EBEI)

The Efficient Binary Execution Indexing (EBEI) used in this thesis is an adaptation of the EPEI by Bin et al., altered to suit three requirements:

1. The index must be generated based on an executable (which may be disassembled), not source code.
2. The index must be collected at runtime rather than relying on prior static analysis.
3. The index must be practical in a reverse engineering context.

It is not necessary to generate a ruleset/EDL describing the full control flow graph, as Bin et al. do. For the purposes of the EBEI it is sufficient to note that there *is* such a language of paths through the graph. To address the constraints of lacking source code and doing all analysis at runtime, this thesis tracks paths in terms of conditional jumps, which may be conceptualized as "rules" which each expand to one of two options (the taken/not taken cases). Because all other operations, including `call`s, `ret`s, and unconditional jumps are deterministic with respect to movements through the control flow graph, a record of decisions made at conditional jumps is sufficient to describe a path. An EBEI is, roughly, a hash of conditional jumps and their decisions; this is described in detail in the Implementation chapter.

However, retaining a hash of *all* conditional jumps encountered would sacrifice the possibility of reconvergence. Instead, like the EPEI, the EBEI maintains a stack. The EBEI's stack, however, is only of function contexts; it is not aware of intra-function control flow contexts. Instead, information about conditional branches is stored per stack frame, and therefore is popped when the stack frame is popped. This means that **the EBEI is calculated only based on branches encountered during functions up to and including the current stack frame** (see Fig. 3-4).

This retains the opportunity for reconvergence (although it may take slightly longer, since state is popped from the stack less often). For example, say that in two different executions, `main` calls `function1`. Assume that the executions behave

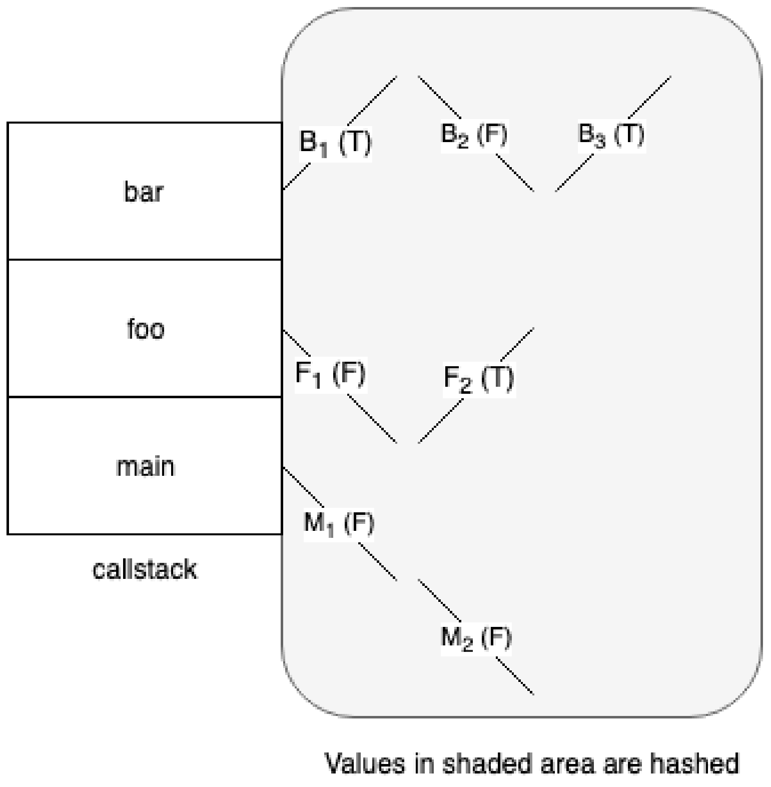


Figure 3-4: The values included in the hash which serves as an execution index

identically within `main` prior to this call, and so the indices are aligned at the moment `function1` is called. If the executions diverge during `function1`, the indices will also diverge. However, when `function1` is popped from the stack, the indices are reset to their value from the moment `function1` was called. As long as the executions continue to behave identically, the indices will stay aligned. If the executions diverge later in `main`, or in another function it calls, this will be detectable in a second divergence of the indices.

Another practical advantage of EBEI over EPEI is that is it less resource-intensive to log. An EBEI is a 32-bit hash, whereas an EPEI may grow very large if many nested contexts are entered, for example in a tight loop (although Bin et al. introduce several optimizations to partially mitigate this). Tracking the EBEI requires tracking state in memory other than a single hash, but this state takes the form of an augmented stack corresponding to the function stack, and so in practice does not grow linearly with instruction count.

# Chapter 4

# Implementation

The trace tool is implemented as a PANDA plugin called `control`. It builds on Brendan Dolan-Gavitt's `callstack_instr` plugin, which implements per-address space call stack tracking, by adding two major capabilities: jump detection and logging the EBEI hash reflecting a subset of execution history (as described in Chapter 3).

## 4.1 State maintained

### 4.1.1 Control flow structure

To aid in generating the index described in Chapter 3, the plugin maintains a data structure containing the desired control flow state. This takes the form of an augmented stack, diagrammed in 4-1. A vector of `stack_entry` structs reflects the function state, with each `stack_entry` pointing to a vector of `Branch` structs (among other fields).

```
struct stack_entry {
    target_ulong pc;   // return address
    std::vector<Branch> control;
    std::size_t hash;
};
```

A `Branch` struct describes an encounter with a conditional jump, recording the program counter of the jump, whether or not it was taken, and how many times:

```
struct Branch{
    target_ulong pc;
    bool taken;
    target_ulong count;
};
```

`stack_entry` structs contain a hash reflecting the up-to-and-including-current-frame control flow state described in the previous chapter. When a new `stack_entry` is pushed, its hash field is initialized from the hash of the previous frame. At every conditional jump encountered, the new branch structure is hashed and XOR'ed with the `.hash` field of the current `stack_entry` (in addition to being appended to the `stack_entry`'s vector of branches):

```
current_stack_entry.hash ^= hashbranch(new_branch);
```

A `stack_entry`'s hash may be altered under two other circumstances: if the user has specified an address range, the hash is reset when that address range is entered to ensure that divergences prior to that point (likely during setup) are ignored. The user may also specify an arbitrary point in execution at which the hash should be reset. These user-controlled resets are roughly analogous to Bin et al's concept of anchor points.

### 4.1.2 Block map

The block map is responsible for maintaining the state necessary to decide whether a given conditional jump was taken or not. It maps a tuple of `(asid, block_start_pc)` to `(jump_pc, landing_pc_if_no_jump)`. In the current implementation only one asid, and therefore one stack, will be used, but indexing by asid is kept for extensibility.

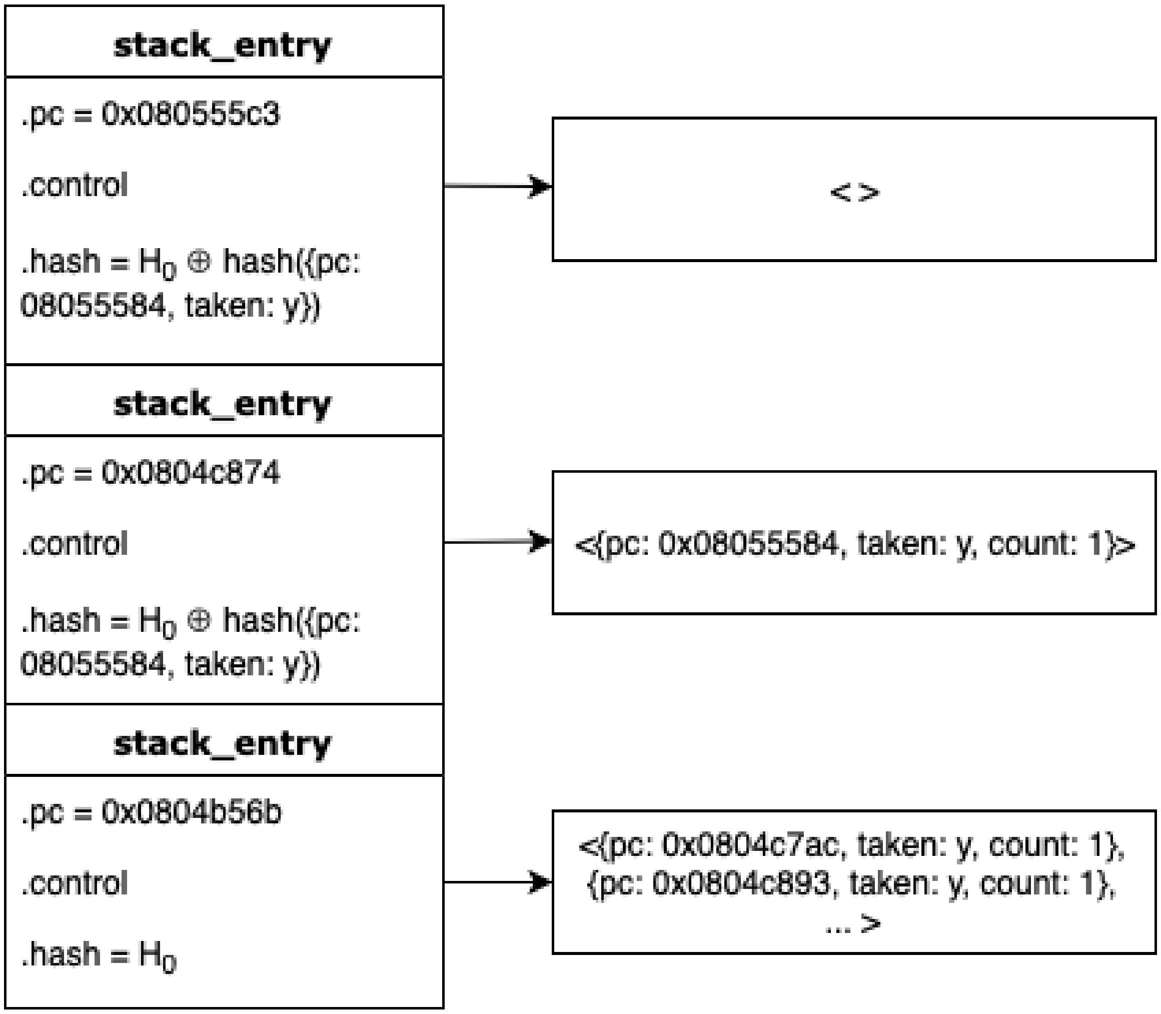

stack_entry
.pc = 0x080555c3
.control
.hash = H0 ⊕ hash({pc: 08055584, taken: y})
stack_entry
.pc = 0x0804c874
.control
.hash = H0 ⊕ hash({pc: 08055584, taken: y})
stack_entry
.pc = 0x0804b56b
.control
.hash = H0
< >
<{pc: 0x08055584, taken: y, count: 1}>
<{pc: 0x0804c7ac, taken: y, count: 1}, {pc: 0x0804c893, taken: y, count: 1}, ... >

Figure 4-1: The control flow structure

The first tuple describes a basic block ending in a conditional jump. The second tuple expresses 1. the pc of that jump, which will be used to construct a Branch object, and 2. the pc of the next sequential block, calculated as the address of the block ending with the jump plus its size. The plugin interacts with this structure according to the following algorithm:

**Algorithm 1** Jump detection algorithm

```
1: procedure jumpDetect
2:     for all $basic\_block$ do
3:         if $last\_block\_jump$ then
4:             $last\_block\_jump = false$
5:             lookup $(asid, last\_start\_pc) \rightarrow (jump\_pc, landing\_pc\_if\_no\_jump)$
6:             $jump\_taken = (landing\_pc\_if\_no\_jump \neq block\_start\_pc)$
7:         disas $basic\_block$
8:         if $basic\_block$ ends in cond. jump then
9:             $last\_block\_jump = true$
10:            $last\_start\_pc = block\_start\_pc$
11:            if $basic\_block$ not in BlockMap then
12:                add $basic\_block$ to BlockMap: $(asid, block\_start\_pc) \rightarrow$
    $(jump\_pc, landing\_pc\_if\_no\_jump)$
```

That is, jumps are detected by having the block ending in a jump set a flag, cache its own start pc, and store a mapping from its start pc to the start pc of the next sequential block. The next block to execute - which may or may not be the next sequential block - then performs a lookup in the block map with the cached pc, yielding the pc of the block that would be next to execute *if* the conditional jump at the end of the block corresponding to the cached pc was not taken. It then simply compares this pc to its own pc. If the values differ, the jump must have been taken.

Under some circumstances, the assumption that the next block to be seen by `after_block_exec` after a not-taken conditional jump will begin at the instruction immediately following the jump instruction will be broken. For example, the first time a floating point instruction is encountered in a given address space, the kernel will take control to set a flag noting that floating point hardware is being used by the process and so floating point state must be cached during context switches. When it returns control to the user code, it will do so at the floating point instruction. This may result in the first part of a block being “skipped” from `after_block_exec`’s perspective, potentially causing a false positive if it happens in the block following a conditional jump that was not taken. This is mitigated by adding state to `before_block_exec` that lets it detect when it is executed twice in a row with no call to `after_block_exec` in between, at which point it sets a flag alerting `after_block_exec` to use the true block start pc which `before_block_exec` has cached for it.

## 4.2 Arguments

The reverse engineer may specify the following values:

- make_hashlog: Specifies whether the running hash should be written to a file at every conditional jump.
- asid: Specifies which address space to track, with blocks from all other address spaces and the kernel being ignored.
- print_at: Specifies the conditional jump at which the full control structure should be logged to a file (intended to be used on a second pass, as explained below).
- start_addr: The beginning of the address range (within the specified asid) of interest, at which the plugin should reset the hash and start logging.
- end_addr: The end of the address range of interest.
- log_all_once_seen: Specifies whether all jumps within the specified address *space* should be logged once the address *range* has been entered, i.e. once the program is executing, whether code from the mmap region should be logged as well.
- force_reset: Allows the user to force a reset of the hash at a particular conditional jump. This is useful if, for example, the executions diverge early in the

context of a given function, but the reverse engineer suspects they will reconverge but then diverge again in an interesting way later in the same function context or higher on the stack.

## 4.3 First pass: Recording control flow hash

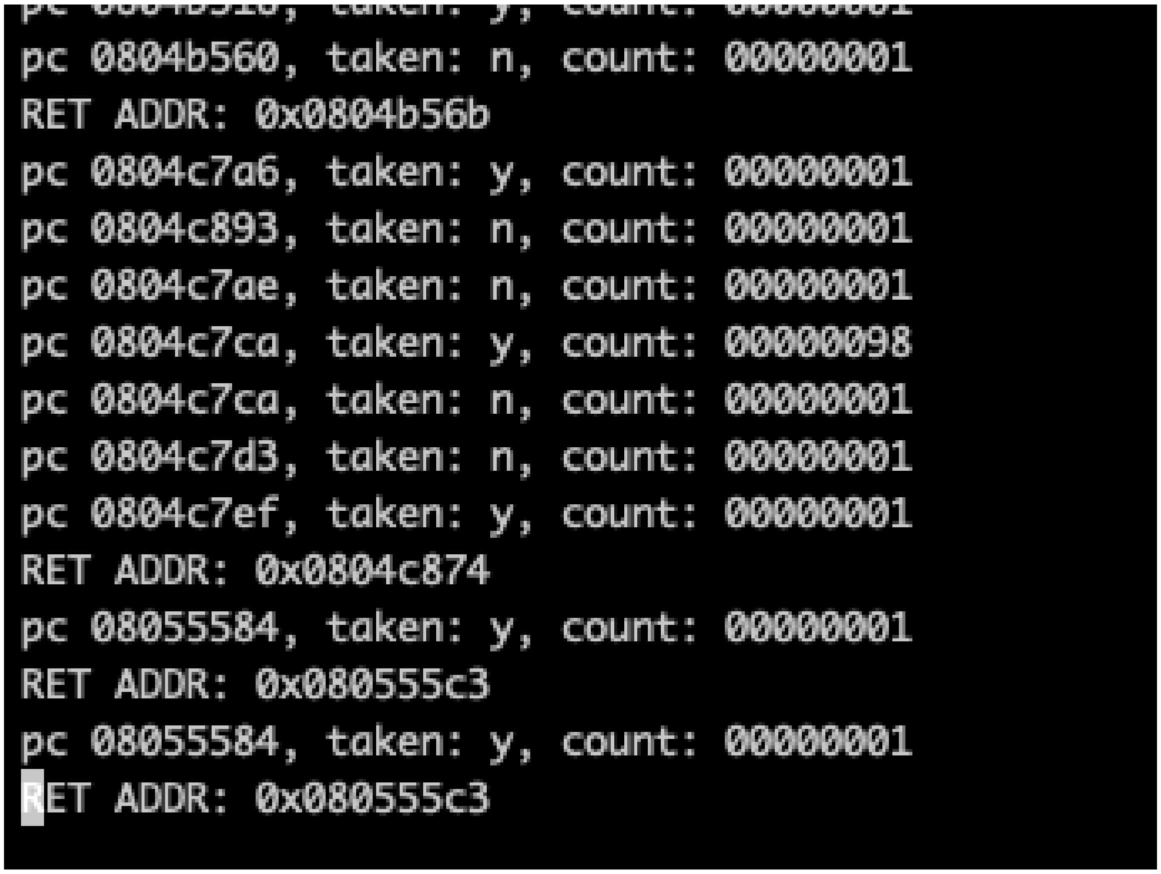
In the expected workflow, the first run of the plugin is responsible for generating a file containing a sequence of hashes. One hash is recorded per conditional jump encountered (that of the top `stack_entry`). This hash reflects the current state of the control flow structure and is updated at every conditional jump.

## 4.4 Second pass: Recording full control flow structure

Because exactly one hash is recorded per conditional jump (within the code of interest), a divergence on the $n$th line of the hash log implies a divergence at the $n$th jump encountered within that same chunk of code. The reverse engineer can use tools such as `vimdiff` to identify line numbers of interest.

That number is then passed to the plugin as the `print_at` argument. At the jump of interest, the full control flow structure is written to a file in a human-readable format, including both return addresses and branches. An example of this structure is pictured in Figure 4-2. The return addresses have the same meaning as in a conventional function stack, i.e. it is the address immediately following the call to the relevant function. The sequences of branches describe control flow within the function which returns to the `RET ADDR` line above them, from the moment that scope was entered to the moment the next function on the stack was called.

In Figure 4-2, for example, the function which returns to `0x804b56b` was entered, three different branches were encountered, the same branch was taken 98 times and then not taken once, two more branches were encountered, and then a function was called (with the return address `0x0804c874`). Within this function, one branch was



```
pc 0804b560, taken: n, count: 00000001
RET ADDR: 0x0804b56b
pc 0804c7a6, taken: y, count: 00000001
pc 0804c893, taken: n, count: 00000001
pc 0804c7ae, taken: n, count: 00000001
pc 0804c7ca, taken: y, count: 00000098
pc 0804c7ca, taken: n, count: 00000001
pc 0804c7d3, taken: n, count: 00000001
pc 0804c7ef, taken: y, count: 00000001
RET ADDR: 0x0804c874
pc 08055584, taken: y, count: 00000001
RET ADDR: 0x080555c3
pc 08055584, taken: y, count: 00000001
RET ADDR: 0x080555c3
```

Figure 4-2: The formatted control flow context displayed to the reverse engineer on a second pass

taken, then another function was called, which also took one branch and called another function. At the moment the stack was printed, the function at the top of the stack has not yet encountered any conditional branches. (Note that at any point during that sequence, a function could have been entered and subsequently returned from without leaving any record in this structure.)

## 4.5 Misc. implementation details

### 4.5.1 Disassembly and jump identification

Basic blocks are disassembled with Capstone and filtered by `cs_insn_group` into blocks ending with calls, returns, and jumps.[2][3] As in `callstack_instr`, instruction sets must be handled separately to distinguish conditional from unconditional jumps.

### 4.5.2 Hashing

The hash used is the RS hash function, modified by Arash Partow.[1][9]. This hash performs well in dealing with low entropy - it is called on the conditional jump program counters, which have low entropy among themselves.

```
inline target_ulong hashpc(target_ulong pc) {
   char buf[sizeof(target_ulong)*2+1];
   int length = sprintf(buf, "" TARGET_FMT_lx "", pc);
   char * str = buf;
   unsigned int b    = 378551;
   unsigned int a    = 63689;
   unsigned int hash = 0;
   unsigned int i    = 0;

   for (i = 0; i < length; ++i) {

      hash = hash * a + (*str);
      a    = a * b;
      str++;
   }
   return (target_ulong)hash;
}
```

Listing 4.1: The modified RS hash used to hash branch program counters

This hash function is a helper to `hashbranch`, below, which generates a hash of an entire branch structure. It first determines which program counter should be hashed. This will not be the exact pc of the conditional jump if the jump is in the mmap region (see lines 4-5 of Listing 4.2 and explanation in Section 4.5.3). The hash of the program counter is then XOR'ed with a constant value if the branch was taken. The resulting value is rotated left by one bit and returned.

[1]https://www.partow.net/programming/hashfunctions/#AvailableHashFunctions

```cpp
inline std::size_t hashbranch(Branch branch) {
    unsigned long pc_to_hash = (unsigned long) branch.pc;
    if (is_libc(branch.pc)) {
        pc_to_hash = branch.pc - libc_base;
    }
    target_ulong result = hashpc(pc_to_hash);
    if (branch.taken) result ^= 0x9e3779b9;
    return (result << 1) | (result >> (sizeof(target_ulong)*CHAR_BIT-1));
}
```

Listing 4.2: The function combining the fields of the Branch structure

The constant `0x9e3779b9`, derived from the golden ratio[2], is the same "random" value as is used in the C++ `boost::hash_combine` function. The result is rotated by 1 to mitigate the possibility of the same branch (i.e. with the same program counter and taken/not taken value) being XOR'ed twice and canceling itself out.

While some warning signs of collisions are observable in the hash logs (see 5-2), this approach performs well enough for useful analysis. For larger recordings it may be necessary to swap in a more heavyweight hash function at the cost of performance. Hashing in the context of describing control flow is known to be a difficult problem due to the low entropy in practice among program counters relative to the theoretical size of the domain.[6]

### 4.5.3 mmap region offset

If control flow within the mmap region is tracked (by either not specifying an address range or setting `log_all_once_seen` to `true`), ASLR basing libc at different addresses will cause spurious divergences. To mitigate this, the first address encountered in the mmap region is saved and all subsequent mmap addresses are hashed relative to it (see 4.2). It is possible for this value to underflow and/or collide with virtual addresses being used by the user code, but given the nature of the analysis this is not disruptive. The analysis will be impacted, however, if two or more separate regions of code are mmapped differently across executions. This is discussed in Future Work.

[2]http://burtleburtle.net/bob/hash/doobs.html

## 4.6 Optimizations

### 4.6.1 Branch counts

Rather than potentially appending many identical branches if a tight loop causes a basic block to be executed repeatedly, a `count` field is present in the `Branch` struct. If the branch about to be appended to the vector is identical to the most recent one - i.e. the pc and "decision" are the same - the most recent branch's `count` field is incremented instead.

### 4.6.2 Exec vs. translate

The most expensive operation done by the plugin is disassembling basic blocks in order to identify blocks ending in conditional jumps. To avoid disassembling a given block every time it executes, the disassembly is done in `after_block_translate` rather than with most of the other plugin code in `after_block_exec`. A mapping of block pc to block type (i.e. whether it ends in a `call`, `ret`, or jump) is maintained, in which `after_block_exec` can perform lookups to determine how to handle the current block.

### 4.6.3 Unconditional jumps

Since unconditional jumps are deterministic, it is not necessary to analyze them even though they impact control flow. At present filtering out unconditional jumps (`jmp`) is only implemented for x86. Including unconditional jumps in the analysis does not break the correctness, since they will simply appear as a jump taken every time and never cause divergence.

# Chapter 5

# Evaluation

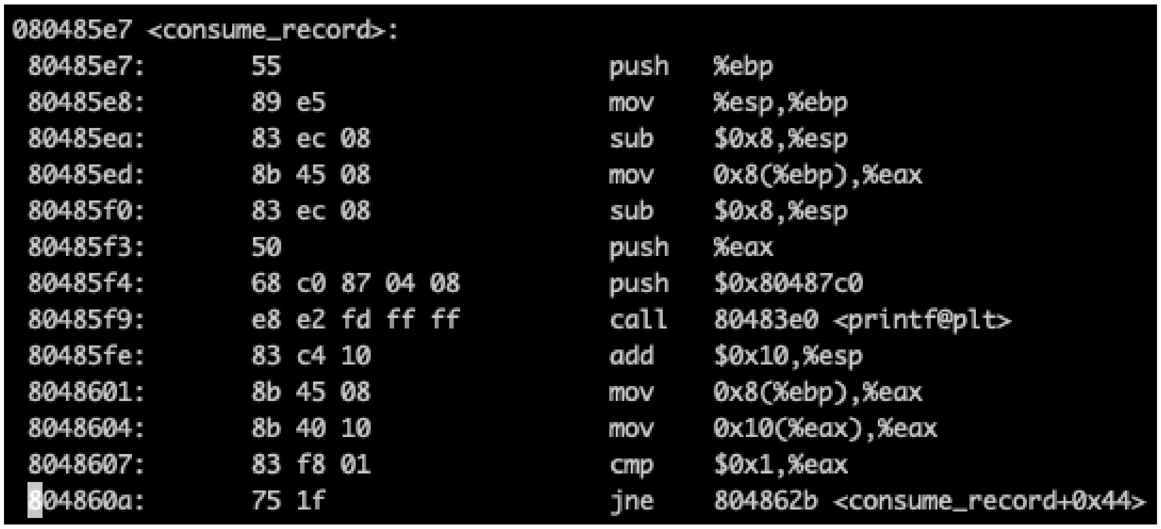
Two test cases are described in this chapter. The first is small and offers easily interpretable results. The second is larger and demonstrates that the `control` plugin is able to filter a large amount of data into a number of "interesting" jumps a human reverse engineer could reasonably look at closely.

In both cases, first two PANDA recordings are created. These recordings are of identical binaries running on different inputs which cause different behavior to be exhibited. These recordings are replayed with the `control` plugin, generating a hashlog (file recording EBEIs) for each. The hashlogs are compared using a tool such as `vimdiff`. Where divergences are observed, `control` is re-run and passed the line number of the divergence, yielding a human-readable record of the control flow state at the moment of divergence.

## 5.1 Text parsing with toy.c

`toy.c`, reproduced in A.2, is a small C test program written by Brendan Dolan-Gavitt.[1] Compiled, it is 396 lines of assembly. It parses a simple binary file, consuming records that may be one of two types. Type-1 records have a 1 in their `type` field and their data is parsed as an integer. Type-2 records have a 2 in their `type` field and their data is parsed as a float. Example binaries with record types 1, 2, 1 (A.3)

[1]https://gist.github.com/moyix/93cd687fde9fb965cfb7d508118d27c1



```
080485e7 <consume_record>:
 80485e7:       55                      push   %ebp
 80485e8:       89 e5                   mov    %esp,%ebp
 80485ea:       83 ec 08                sub    $0x8,%esp
 80485ed:       8b 45 08                mov    0x8(%ebp),%eax
 80485f0:       83 ec 08                sub    $0x8,%esp
 80485f3:       50                      push   %eax
 80485f4:       68 c0 87 04 08          push   $0x80487c0
 80485f9:       e8 e2 fd ff ff          call   80483e0 <printf@plt>
 80485fe:       83 c4 10                add    $0x10,%esp
 8048601:       8b 45 08                mov    0x8(%ebp),%eax
 8048604:       8b 40 10                mov    0x10(%eax),%eax
 8048607:       83 f8 01                cmp    $0x1,%eax
 804860a:       75 1f                   jne    804862b <consume_record+0x44>
```

Figure 5-1: Assembly of `consume_record` from `toy.c`

and 2, 2, 2 (A.4) are reproduced in the appendix.

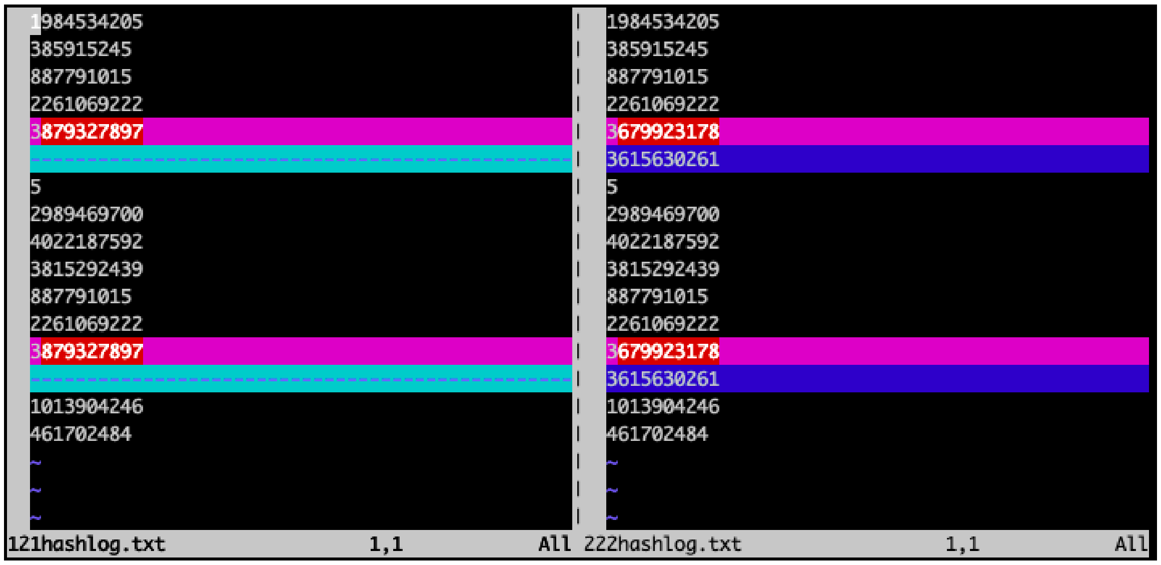
The initial check of whether a record is type-1 (seen in line 36 of the source) compiles to the assembly shown in Figure 5-1.

Two recordings are made of `toy.c` executing, one with the `121` binary and one with the `222` binary. First the `asidstory` plugin is used to learn the asid for `toy.c` in each recording. Then the `control` plugin is run on each recording, generating a hashlog apiece. (The presence of an existing hashlog is not checked, so the first hashlog generated must be renamed before generating the second.) The hashlogs are compared with `vimdiff`, as shown in Figure 5-2.

This reveals that divergences happened on lines 5 and 12, i.e. the fifth and twelfth conditional jump encountered by the `121` recording. Note that either recording can be used during the second pass, but the line numbers passed must correspond to the line numbers of the divergences in *that* hashlog - if the `222` recording is used on a second pass instead, the correct line numbers would be 5 and 13.

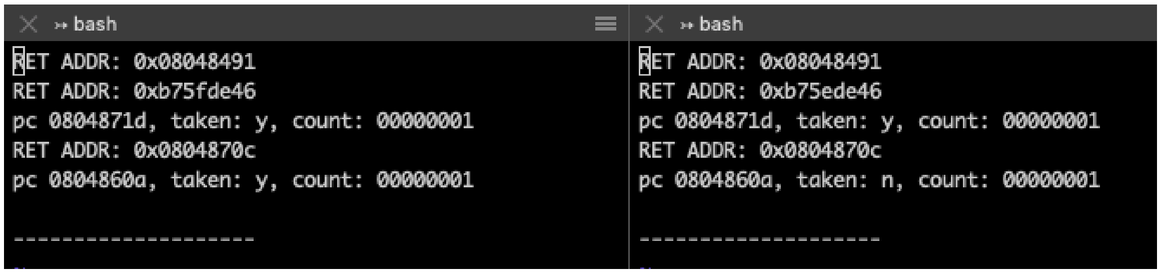
The recordings are run again, this time with the `print_at` argument set to 5. In principle it is not necessary to run the second pass on both recordings, since they should always be identical up to the last branch, at which they should diverge. These commands generate the records seen in Figure 5-3 in `controllog_readable.txt`.

Sure enough, the recordings are in step until they reach the conditional jump at



```
1984534205                                    | 1984534205
385915245                                     | 385915245
887791015                                     | 887791015
2261069222                                    | 2261069222
3879327897                                    | 3679923178
----------------------------------------------| 3615630261
5                                             | 5
2989469700                                    | 2989469700
4022187592                                    | 4022187592
3815292439                                    | 3815292439
887791015                                     | 887791015
2261069222                                    | 2261069222
3879327897                                    | 3679923178
----------------------------------------------| 3615630261
1013904246                                    | 1013904246
461702484                                     | 461702484
~                                             | ~
~                                             | ~
~                                             | ~
121hashlog.txt                1,1          All  222hashlog.txt                1,1          All
```

Figure 5-2: The diff between the hashlogs of two executions of `toy.c`



```
bash                                          bash
RET ADDR: 0x08048491                          RET ADDR: 0x08048491
RET ADDR: 0xb75fde46                          RET ADDR: 0xb75ede46
pc 0804871d, taken: y, count: 00000001        pc 0804871d, taken: y, count: 00000001
RET ADDR: 0x0804870c                          RET ADDR: 0x0804870c
pc 0804860a, taken: y, count: 00000001        pc 0804860a, taken: n, count: 00000001

--------------------                          --------------------
```

Figure 5-3: The full control context at the moment of divergence between two executions of `toy.c`

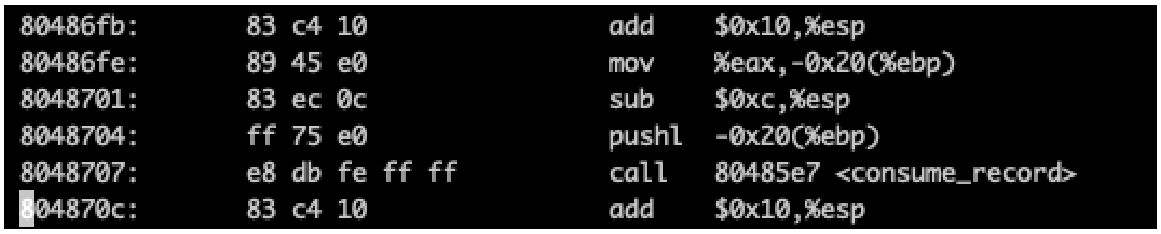

```
80486fb:      83 c4 10            add    $0x10,%esp
80486fe:      89 45 e0            mov    %eax,-0x20(%ebp)
8048701:      83 ec 0c            sub    $0xc,%esp
8048704:      ff 75 e0            pushl  -0x20(%ebp)
8048707:      e8 db fe ff ff      call   80485e7 <consume_record>
804870c:      83 c4 10            add    $0x10,%esp
```

Figure 5-4: The context of the call to `consume_record` in the assembly of `toy.c`

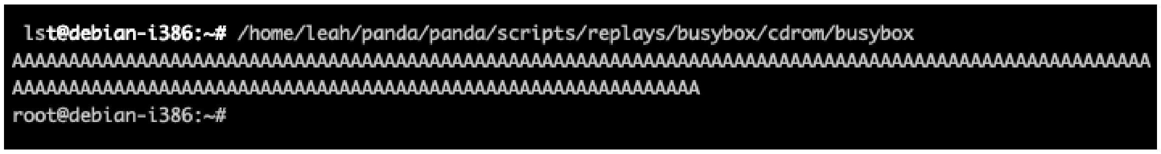
`0x804860a`, which is conditional on a check of some value (which on inspection is indeed the record type) against a constant 1.

This check is made following the `retaddr` value `0x804870c`, i.e. this is the location to which the current function will return. The address `0x804870c` is located in `main`, immediately following a call to `consume_record`, consistent with the hypothesis that the different record type is the cause of the divergence. The disassembly of this region is shown in Figure 5-4

Note that by the time `consume_record` is called, `main` has already called and returned from `parse_header`, `parse_record`, and several libc functions. They do not appear in `controllog_readable.txt` because they have already been popped from the stack. This is how it is possible for only two branches to appear on the stack despite the plugin being set to print at the fifth conditional jump encountered - in this case, jumps encountered in libc are not considered, so the "missing" jumps must be located in `parse_header` or `parse_record` (and some investigation and printing reveals that this is indeed the case).

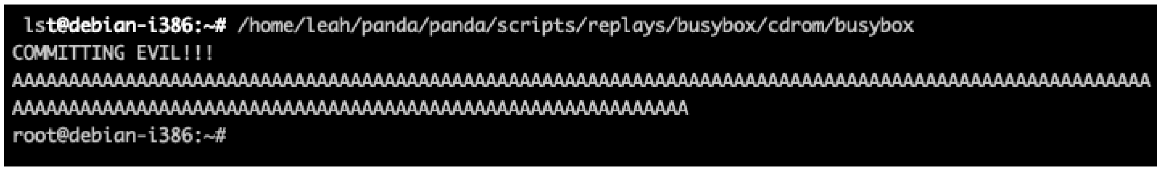
## 5.2 Modified BusyBox

BusyBox is an open-source resource that "combines tiny versions of many common UNIX utilities into a single small executable", intended for embedded Linux applications.[1] As a larger test, a statically-linked, 32-bit BusyBox binary was compiled from source, yielding 483,762 lines of assembly. The source code for `ls` was slightly modified, with the last line of Listing 5.1 added.



```
 lst@debian-i386:~# /home/leah/panda/panda/scripts/replays/busybox/cdrom/busybox
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
root@debian-i386:~#
```

Figure 5-5: The output of `busybox ls` with a non-“trigger” file present



```
 lst@debian-i386:~# /home/leah/panda/panda/scripts/replays/busybox/cdrom/busybox
COMMITTING EVIL!!!
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
root@debian-i386:~#
```

Figure 5-6: The output of `busybox ls` with a “trigger” file present

```c
/* find the longest file name, use that as the column width */
                for (i = 0; dn[i]; i++) {
                        int len = calc_name_len(dn[i]->name);
                        if (column_width < len)
                                column_width = len;

                }
                if (column_width == 0xa1) printf("COMMITTING EVIL!!!\n")
                    ;
```

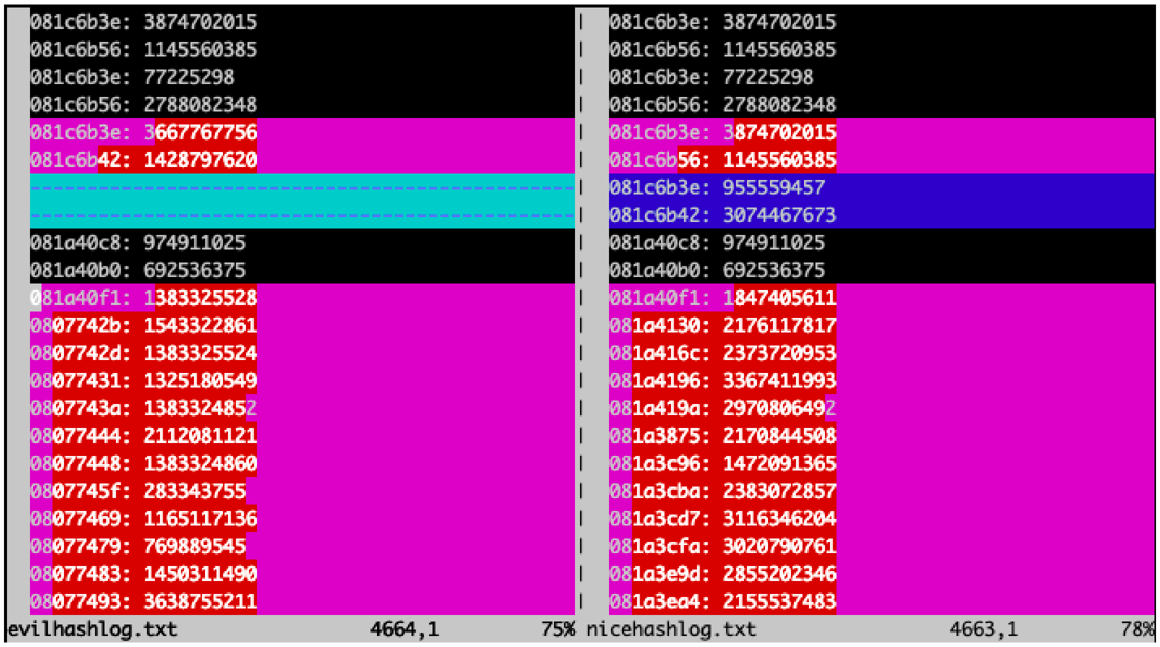
Listing 5.1: ls.c

As the `column_width`, i.e. the length of the longest item name is being calculated, the code exhibits a “misbehavior” under some specific circumstance - in this test example, it prints a message if a file whose name is exactly 161 (`0xa1`) characters long is present. A recording of this modified BusyBox running `ls` was run in an environment where only a length-162 filename was present (see Figure 5-5) and one where only a length-161 filename was present (see Figure 5-6).

The same procedure as the `toy.c` test is followed. A log of 6182 hashes is generated, with 8 divergences present between the two recordings. One of the divergences is the one shown in Figure 5-7, with the length-161 trigger for the “malicious” code present on the left and a length-162 non-trigger on the right.

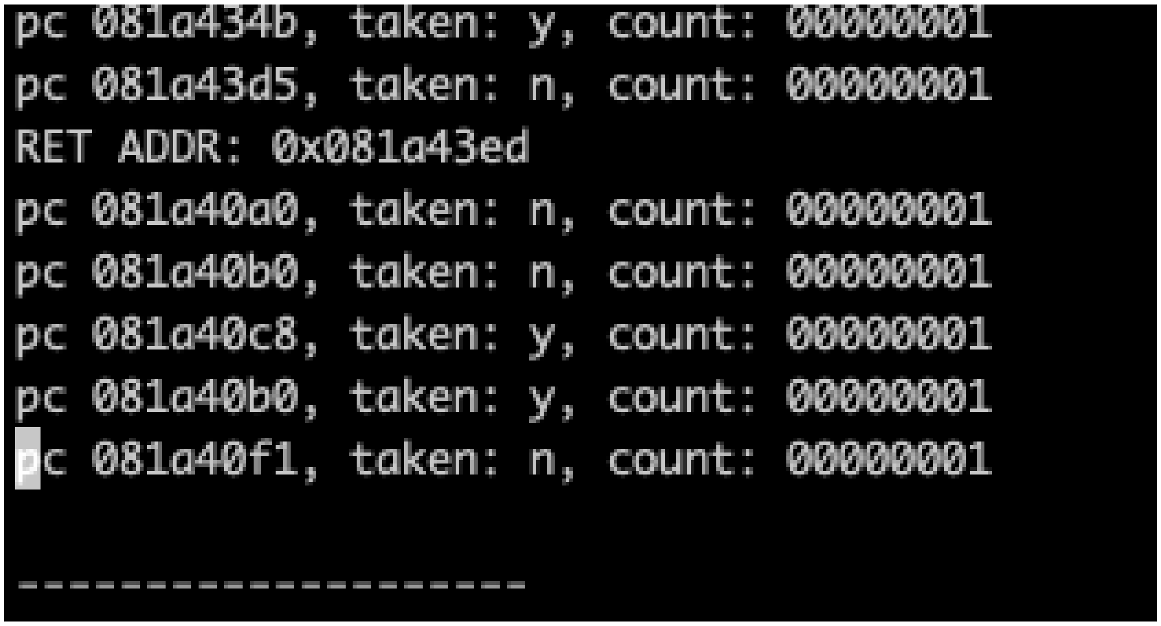
A second pass is done, printing the full structure at the 4664th line. A 184-line augmented stack is generated, terminating in the sequence shown in 5-8.

The area of the BusyBox code surrounding the jump at `0x81a40f1` is shown in 5-9.



```
081c6b3e: 3874702015                        | 081c6b3e: 3874702015
081c6b56: 1145560385                        | 081c6b56: 1145560385
081c6b3e: 77225298                          | 081c6b3e: 77225298
081c6b56: 2788082348                        | 081c6b56: 2788082348
081c6b3e: 3667767756                        | 081c6b3e: 3874702015
081c6b42: 1428797620                        | 081c6b56: 1145560385
--------------------------------------------| 081c6b3e: 955559457
--------------------------------------------| 081c6b42: 3074467673
081a40c8: 974911025                         | 081a40c8: 974911025
081a40b0: 692536375                         | 081a40b0: 692536375
081a40f1: 1383325528                        | 081a40f1: 1847405611
0807742b: 1543322861                        | 081a4130: 2176117817
0807742d: 1383325524                        | 081a416c: 2373720953
08077431: 1325180549                        | 081a4196: 3367411993
0807743a: 1383324852                        | 081a419a: 2970806492
08077444: 2112081121                        | 081a3875: 2170844508
08077448: 1383324860                        | 081a3c96: 1472091365
0807745f: 283343755                         | 081a3cba: 2383072857
08077469: 1165117136                        | 081a3cd7: 3116346204
08077479: 769889545                         | 081a3cfa: 3020790761
08077483: 1450311490                        | 081a3e9d: 2855202346
08077493: 3638755211                        | 081a3ea4: 2155537483
evilhashlog.txt           4664,1        75% nicehashlog.txt           4663,1        78%
```

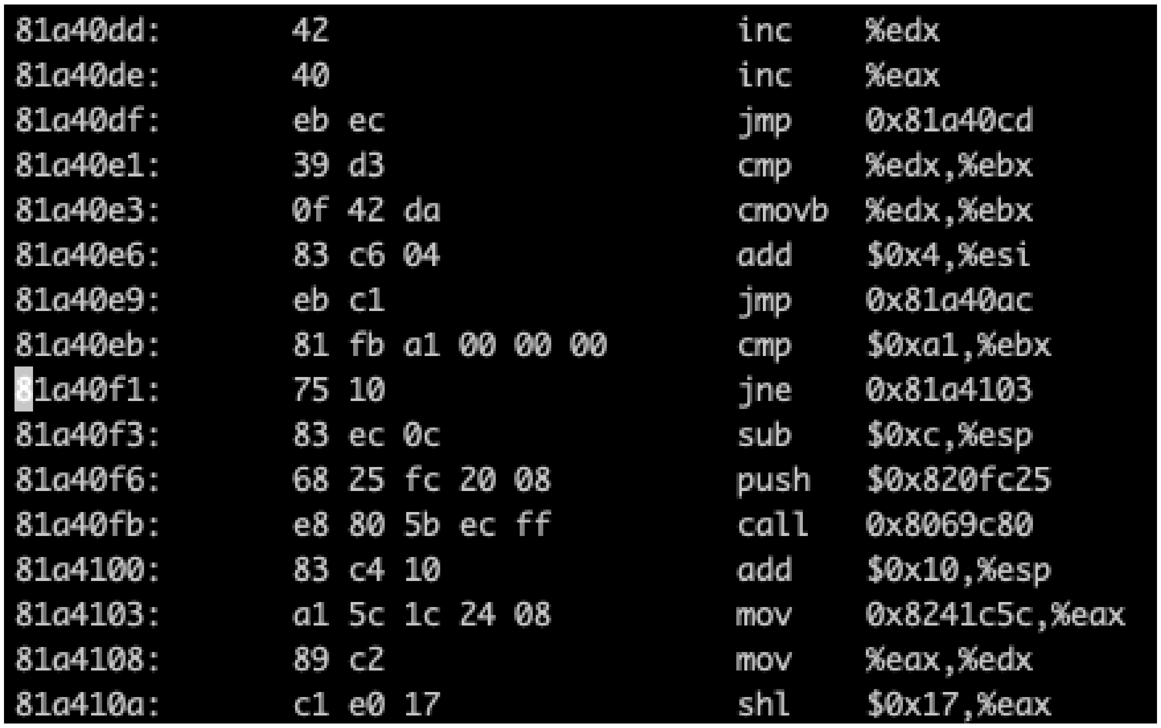
Figure 5-7: The diff between the hashlogs of two executions of `busybox ls`



```
pc 081a434b, taken: y, count: 00000001
pc 081a43d5, taken: n, count: 00000001
RET ADDR: 0x081a43ed
pc 081a40a0, taken: n, count: 00000001
pc 081a40b0, taken: n, count: 00000001
pc 081a40c8, taken: y, count: 00000001
pc 081a40b0, taken: y, count: 00000001
pc 081a40f1, taken: n, count: 00000001


--------------------
```

Figure 5-8: The end of the full control context at the moment of divergence between two executions of `busybox ls`



```
81a40dd:      42                      inc    %edx
81a40de:      40                      inc    %eax
81a40df:      eb ec                   jmp    0x81a40cd
81a40e1:      39 d3                   cmp    %edx,%ebx
81a40e3:      0f 42 da                cmovb  %edx,%ebx
81a40e6:      83 c6 04                add    $0x4,%esi
81a40e9:      eb c1                   jmp    0x81a40ac
81a40eb:      81 fb a1 00 00 00       cmp    $0xa1,%ebx
81a40f1:      75 10                   jne    0x81a4103
81a40f3:      83 ec 0c                sub    $0xc,%esp
81a40f6:      68 25 fc 20 08          push   $0x820fc25
81a40fb:      e8 80 5b ec ff          call   0x8069c80
81a4100:      83 c4 10                add    $0x10,%esp
81a4103:      a1 5c 1c 24 08          mov    0x8241c5c,%eax
81a4108:      89 c2                   mov    %eax,%edx
81a410a:      c1 e0 17                shl    $0x17,%eax
```

Figure 5-9: The context of the diverging jump in the assembly of `busybox`

It immediately follows a comparison to the constant `0xa1` - the only comparison to this constant made in the program, a good sign that this is indeed the location at which the different input causes the divergence of interest. The divergence at line 4661 is attributable to the `for` loop (included in the snippet above) taking one extra iteration to count the longer filename. Note that the two extra jumps are a repetition of the same sequence of two that precedes the divergence. The remaining false positives are attributable to earlier processing of the filename, waiting on locks, and hash collisions - if `vimdiff` erroneously aligns two hashes due to a collision, another pseudo-divergence is created later in the file when the true realignment is found (as a chunk of one log has been shifted relative to the other).

While several false positives are generated, the plugin is successful in narrowing over 6,000 conditional jumps down to 8, significantly reducing the search space for the reverse engineer. It imposes a slowdown of about a factor of 2 (from 0.6 to 1.2 seconds to replay the BusyBox recording).

# Chapter 6

# Conclusion

This thesis demonstrates the viability of trace alignment without source code using PANDA. By defining an execution index that is sensitive to structural context, but only that contained in the current function and lower on the stack, it is possible to detect multiple meaningful divergences in execution. This index is logged on a first pass. Line numbers in this log correspond one-to-one with conditional jumps encountered by the program, so on a second pass the line number of a divergence can be passed as an argument and the full context at that point can be logged in a human-readable way.

## 6.1 Future Work

### 6.1.1 Analysis

#### Structural Indexing

Possible future work would include more precisely implementing the structural index tracking described by Bin et al. at runtime. This would require correlating basic blocks to their location in the source code structure, and in turn tracking location in structures such as loops and if/then blocks at runtime. One possible approach is enumerating common patterns in how source code control structures are converted to assembly and maintaining state that would allow the identification of these structures.

While it may not be feasible in theory to capture every possible mapping between source code structure and runtime control flow, in practice optimized code tends to yield a "restricted subclass" of control flow graphs and so this analysis may be feasible.[8]

### Multithreading

The `control` plugin does not have a concept of multithreading, i.e. multiple threads of execution taking place in the same address space. The PANDA API does not currently offer a way to distinguish different processes running within the same address space (i.e. generated with `clone`). Some PANDA plugins, such as `callstack_instr`, use heuristics to determine which of multiple processes with the same asid a given basic block is coming from. Future work could include increasing the robustness of this or building the functionality into PANDA itself (or QEMU).

### Handling mmap and JIT compilation

More detailed treatment of the mmap region is likely to be necessary for useful analysis of JIT-compiled code. If the number of runtime-determined code regions is small, it should be reasonably practical for a reverse engineer to make small changes to the plugin and hardcode in the address ranges. Handling many small chunks of code being run from the heap is more difficult and may require dropping the pc entirely as a definitive identifier of a given conditional jump.

### Hash selection

Since hash functions have different characteristics and tradeoffs, it would be useful to let the reverse engineer specify one as an argument, depending on their priorities (speed, minimizing false reconvergences, etc.).

### 6.1.2 Usability

There are several user-friendliness improvements that could be made to the `control` plugin. In particular, several steps of the workflow are currently done manually but could be automated:

- **`objdump` parsing:** All interaction with the disassembly of the binary is currently carried out via the user. Some of this - for example, getting the address range and checking the line before a return address to learn what function has just been returned from - could be automated.

- **Automated interaction with taint analysis:** Once the reverse engineer has identified a conditional jump of interest, a likely next step is to track the flag influencing that jump back to some raw input. This interaction between `control` and other tools may be scriptable.

# Appendix A

# Code Listings

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>

#pragma pack(1)
#define MAGIC 0x4c415641

enum {
    TYPEA = 1,
    TYPEB = 2
};

typedef struct {
uint32_t magic;     // Magic value
uint32_t reserved;  // Reserved for future use
uint16_t num_recs;  // How many entries?
uint16_t flags;     // None used yet
uint32_t timestamp; // Unix Time
} file_header;

typedef struct {
  char bar[16];
  uint32_t type;
  union {
  float fdata;
  uint32_t intdata;
      } data;
} file_entry;

void parse_header(FILE *f, file_header *hdr) {
  if (1 != fread(hdr, sizeof(file_header), 1, f))
    exit(1);
  if (hdr->magic != MAGIC)
    exit(1);
}
```

Listing A.1: toy.c

```

file_entry * parse_record(FILE *f) {
    file_entry *ret = (file_entry *) malloc(sizeof(file_entry));
  if (1 != fread(ret, sizeof(file_entry), 1, f))
    exit(1);
  return ret;
}

void consume_record(file_entry *ent) {
  printf("Entry: bar = %s, ", ent->bar);
  if (ent->type == TYPEA) {
    printf("fdata = %f\n", ent->data.fdata);
  }
  else if (ent->type == TYPEB) {
    printf("intdata = %u\n", ent->data.intdata);
  }
  else {
    printf("Unknown type %x\n", ent->type);
    exit(1);
  }
  free(ent);
}

int main(int argc, char **argv) {
  FILE *f = fopen(argv[1], "rb");
    file_header head;
  parse_header(f, &head);
  printf("File timestamp: %u\n", head.timestamp);
  unsigned i;
  for (i = 0; i < head.num_recs; i++) {
        file_entry *ent = parse_record(f);
    consume_record(ent);
    }
  return 0;
}
```

Listing A.2: toy.c

```
00000000: 4156 414c 0000 0000 0300 0000 9915 8057  AVAL...........W
00000010: 6865 6c6c 6f00 0000 0000 0000 0000 0000  hello...........
00000020: 0100 0000 4141 4840 676f 6f64 6279 6500  ....AAH@goodbye.
00000030: 0000 0000 0000 0000 0200 0000 2a00 0000  ............*...
00000040: 6575 6c65 7200 0000 0000 0000 0000 0000  euler...........
00000050: 0100 0000 4141 2d40 0a                   ....AA-@.
```

Listing A.3: testbin121

```
00000000: 4156 414c 0000 0000 0300 0000 9915 8057  AVAL...........W
00000010: 6865 6c6c 6f00 0000 0000 0000 0000 0000  hello...........
00000020: 0200 0000 4141 4840 676f 6f64 6279 6500  ....AAH@goodbye.
00000030: 0000 0000 0000 0000 0200 0000 2a00 0000  ............*...
00000040: 6575 6c65 7200 0000 0000 0000 0000 0000  euler...........
00000050: 0200 0000 4141 2d40 0a                   ....AA-@.
```

Listing A.4: testbin222

# Bibliography

[1] Busybox: The swiss army knife of embedded linux. `https://busybox.net/about.html`.

[2] The capstone disassembly framework. `http://www.capstone-engine.org/documentation.html`.

[3] Function capstone_sys::cs_insn_group. `https://docs.rs/capstone-sys/0.2.0/capstone_sys/fn.cs_insn_group.html`.

[4] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC, pages 41–41, Berkely, CA, USA, 2005. USENIX Association.

[5] Brendan Dolan-Gavitt, Josh Hodosh, Patrick Hulin, Tim Leek, and Ryan Whelan. Repeatable reverse engineering with panda. In *Proceedings of the 5th Program Protection and Reverse Engineering Workshop*, PPREW-5, pages 4:1–4:11, New York, NY, USA, 2015. ACM.

[6] Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuwen Tu, Kang Li, Zhongyu Pei, and Zuoning Chen. Collafl: Path sensitive fuzzing. *IEEE Symposium on Security and Privacy*, 2018.

[7] Noah M. Johnson, Juan Caballero, Kevin Zhijie Chen, Stephen McCamant, Pongsin Poosankam, Daniel Reynaud, and Dawn Song. Differential slicing: Identifying causal execution differences for security applications. *2011 IEEE Symposium on Security and Privacy*, 2011.

[8] Ken Kennedy and Linda Zucconi. Applications of a graph grammar for program control flow analysis. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL, pages 72–85, New York, NY, USA, 1977. ACM.

[9] Robert Sedgewick. *Algorithms in C.* August 2001.

[10] Ray Wang. A system for dynamic slicing and program visualization. Master's thesis, Massachusetts Institute of Technology, 2 2019.

[11] Bin Xin, William N. Sumner, and Xiangyu Zhang. Efficient program execution indexing. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, volume 43 of *PLDI*, pages 243–248, New York, NY, USA, 2008. ACM.