



SAPIENZA  
UNIVERSITÀ DI ROMA

## Implementation trade-offs in using Intel Pin for instruction tracing of complex programs

Facoltà di Ingegneria dell'Informazione, Informatica e Statistica  
Corso di Laurea in Ingegneria Informatica

Candidate

Andrea Tulumiero

ID number 1708274

Thesis Advisor

Prof. Camil Demetrescu

Co-Advisors

Dr. Daniele Cono D'Elia

Dr. Emilio Coppa

Academic Year 2017/2018

Thesis not yet defended

---

**Implementation trade-offs in using Intel Pin for instruction tracing of complex programs**

Bachelor thesis. Sapienza – University of Rome

© 2018 Andrea Tulimiero. All rights reserved

This thesis has been typeset by L<sup>A</sup>T<sub>E</sub>X and the Sapthesis class.

Version: July 7, 2018

Author's email: [tulimiero.andrea@gmail.com](mailto:tulimiero.andrea@gmail.com)

*Dedicated to  
the ones who have been by my side  
during this mesmerizing journey*



## Abstract

With the development of increasingly advanced techniques to hide the malicious payload of a Malware, the community of reverse engineers and security researchers has been facing more and more complex programs which brought about the need of more advanced analysis than classic ones based on static code inspection. To truly understand what such malicious programs do, an analyst needs to look at them while they are executing, thus tools to carry out their analyses at runtime have become one of the most powerful weapons to face new threats.

Among the techniques for the design and implementation of such tools there is *dynamic binary instrumentation* (DBI), an advanced solution that makes it possible to instrument a program dynamically (i.e., while it is running), allowing for a fine-grained inspection of its execution. Although this technique is very powerful, it carries with it some performance and accuracy trade-offs. In this thesis we will build tools to record instructions and reconstruct the control flow graph of a possibly malicious program, discussing during the journey the challenges introduced by the usage of DBI and proposing some solutions to mitigate these problems.



# Contents

<b>Introduction</b>	<b>ix</b>
<b>1 Dynamic Binary Instrumentation</b>	<b>1</b>
1.1 Technique Overview . . . . .	1
1.2 Choosing Intel Pin . . . . .	2
1.3 Placing Instrumentation . . . . .	3
<b>2 Tracer overview</b>	<b>5</b>
2.1 Tracing points of interest . . . . .	5
2.2 Multithreaded Programs . . . . .	7
2.3 Challenges in the Analysis . . . . .	8
2.4 CFG Reconstruction . . . . .	9
<b>3 Implementation</b>	<b>13</b>
3.1 Unbuffered version . . . . .	14
3.2 Buffered version . . . . .	14
3.3 Thread buffered version . . . . .	16
3.4 Dump the Code . . . . .	18
<b>4 Evaluation</b>	<b>21</b>
4.1 Methodology . . . . .	21
4.2 Results . . . . .	23
<b>5 Conclusions</b>	<b>27</b>





# Introduction

To analyze some pieces of malicious software – also known as Malware – it might be necessary to carry out a live analysis of the program. This need derives from the fact that some obfuscated Malware samples reveal themselves only while they are running, in order to make it harder for security researchers to understand what they do. So if you look at one such sample as a normal binary file, you would lose a great part of the instructions of the program since they are hidden from the sight.

This brought about the need for exploring Dynamic Binary Instrumentation (DBI), which is a certain type of analysis made while the program is running. It consists of inserting some pieces of code among the instructions of the original program, to gather information about it at runtime. DBI is a commonly used technique among researches nowadays, and one of the tools that let you implement this technique is Intel PIN. The core contribution of the thesis is to understand which are the implementation trade-offs linked to the usage of this technique, in particular of this specific tool, in tracing malicious software and propose some solutions to mitigate them.

One of the situations in which it is crucial to analyze the program while it is running is when it is concealed through *packing*, that is the practice of compressing and/or encoding the binary, and then decoding it while it is running. The real structure of a so-called *packed* program and its instructions are revealed only at runtime.

We build a proof-of-concept tool that reconstructs the CFG of a Windows program through Dynamic Binary Instrumentation using Intel PIN, and we explore whether is it possible or not to do the same for a packed program. For the scope of this thesis, we do not consider advanced packing techniques that use Self-Modifying Code, which is code that patches itself – even within the scope of the current basic block – to change its behavior while it is executing.

## Structure of the thesis

In Chapter 1 we will explain what DBI is and we will take a closer look at Intel PIN and the features it offers. In Chapter 2 we will discuss the overall architecture of our tracer tool and we will highlight the challenges we will be dealing with. In Chapter 4 we will evaluate the results obtained from our tests, comparing the different solutions proposed and implemented in the previous chapters. In Chapter 5 we will sum up what we learned from this experience.



## Chapter 1

# Dynamic Binary Instrumentation

In this chapter, we will understand what Dynamic Binary Instrumentation is and we will explain why we chose Intel Pin for the implementation of our ideas. First, we will see how it works, then we will discuss why we decided to use this technique to build our tool, and then we will take a closer look at how Pin works and how we can use it, to make it easier to understand the design choices we will take in the following chapters.

### 1.1 Technique Overview

In order to understand what Dynamic Binary Instrumentation is, let's break it into pieces and analyze each of them. First of all, what does "Instrumentation" mean? It is the practice of inserting instructions into a program to observe its state while it is running. Several studies can be carried out on a program through instrumentation, to name a few:

- *Performance analysis*: finding out bottlenecks in the execution of a program to try and remove them to make the program run faster.
- *Debugging*: understanding why a program fails in some situations.
- *Tracing*: which consists of tracing what a program does in a general way.

Instrumentation can easily be implemented by adding instructions to the source code of the investigated program and then by re-compiling it. This, however, means that we need to know the source code of the program, and this is not the case in many situations in which reverse engineers and security researchers find themselves.

Here comes the "Dynamic" part. In fact, such a tool gives us the ability to insert our instrumentation code into the program without the need to change the source code or to recompile it.

## 1.2 Choosing Intel Pin

According to the official Intel documentation [7], “Pin is a dynamic binary instrumentation framework for the IA-32, x86-64 and MIC instruction-set architectures that enables the creation of dynamic program analysis tools.” It has been used in several projects from both industry and academia carrying out researches in many fields.


What Pin does is a “just in time” (JIT) compilation of the program, thus it reads the binary, adds the analysis instructions we want, and then compiles it back to execute it. Differently from a normal compilation, as we stated previously, we do not need the source code of the executable.

Now that we know how the engine of Pin works, we need to know how to control this tool. The component we will develop is a so-called Pintool, which is a dynamically loaded library (DLL) that we pass to Pin as a command line argument when we invoke it. Pin then loads the Pintool and the observed program into the same address space, so we have access to all the executable’s data and file descriptors.

Since Pin executes the program’s instructions as they were normally executed without Pin, this gives us the possibility to inspect all the interactions that the program has with the underlying operating system as well. It is possible to write a Pintool in C/C++, which means that writing fairly complex analysis programs should not be prohibitive.

A Pintool is divided mainly into two parts: **analysis** and **instrumentation**. To better understand this core concept we will make an example. Let’s assume we want to count all the instructions of a program. This is how the different jobs are divided in a Pintool used in a tutorial session at a leading programming languages conference [8]:

### Instrumentation vs. Analysis



- Instrumentation routines** define where instrumentation is **inserted**
  - e.g., before instruction
  - ☞ **Occurs *first time* an instruction is executed**
- Analysis routines** define what to do when instrumentation is **activated**
  - e.g., increment counter
  - ☞ **Occurs *every time* an instruction is executed**

12

Pin PLDI Tutorial 2007

Figure 1.1. The analysis and instrumentation parts of a Pintool.

And this is what the program will look like after the dynamic instrumentation has taken place:

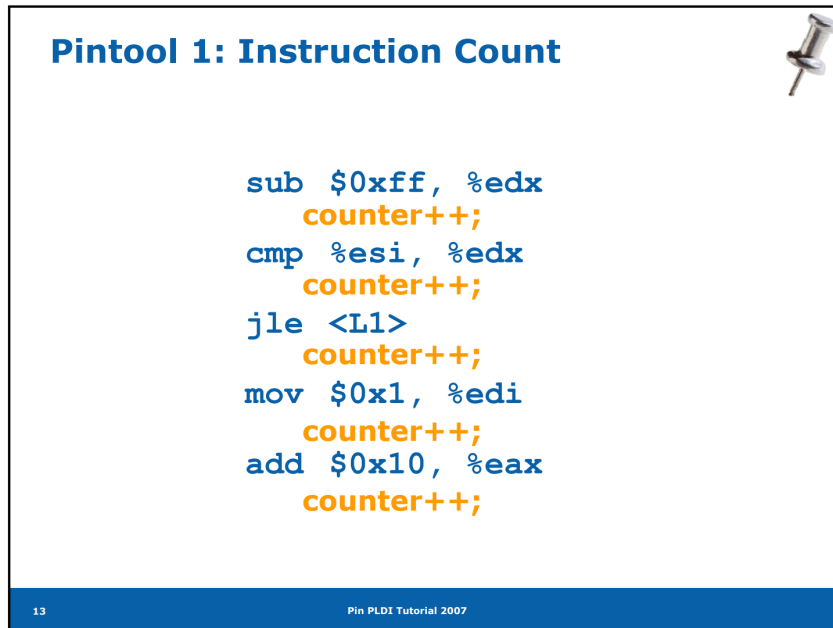


Figure 1.2. All the instructions that will be executed.

### 1.3 Placing Instrumentation

So far we have seen that with Pin it is possible to insert pieces of code to do almost whatever analysis we want. We will now examine the **instrumentation** phase of a Pintool, which is the positioning of the instrumentation code inside the analyzed program. Pin offers various strategies to locate the right spot to insert the instrumentation code. The idea is that we can specify a callback<sup>1</sup> to be fired when a precise event occurs, such as a load of an Image. We will follow a top-down approach from the least to the most fine-grained alternative we have to specify the callback calling event:

- *Image*: with this, we are able to inspect the whole Image when it gets loaded, and we are able to get some useful information from this like the High and Low address where the image was loaded.
- *Section*: allows us to walk through the different sections of an Image.
- *Routine*: as the name might suggest, Pin will fire our callback every time a routine (i.e., function) of the program is encountered. Although this is a powerful instrumentation technique, we would still have too little information to be able to reconstruct the Basic Blocks in it.
- *Trace*: a trace is a sequence of instructions which Pin guarantees are entered only at the top, but may contain different exits. This is because a Trace

<sup>1</sup>A callback is just a function that we want to be called when some sort of event occurs.

is commonly composed of different Basic Blocks (BBLs), which are single-entrance, single-exit sequence of instructions.

- *Instruction*: which is just a convenient way offered by Pin to loop through all the instructions of a program without the need to iterate over the traces, the BBLs, and the instructions.

For the tracing part of our tool, we decided to instrument at *Instruction* level, the most fine-grained of the ones we mentioned earlier. The reason we opted for this is discussed in section 2.1. We will also exploit the *Image* and *Section* to gather some useful one-time information of the program.



The assumptions we previously made, however, are true only if the program we are instrumenting does not present Self-Modifying Code (SMC). This type of software modifies its own instructions while it is running, often within the scope of the currently executing basic block. So while a packed program works like a kind of self-contained zipped file in which there is the compressed code and a stub to decompress it, a self-modifying program gives new meaning to its instructions modifying some bytes of its own code. To analyze such a program we would need to undertake a much more complicated and onerous analysis that requires the inspection and recording of all instructions. Furthermore, these programs are not very common, so we will not consider them while building our tracer.

As mentioned earlier, to reconstruct the path followed by the program, we need to collect information regarding points where jump instructions take place, and the target of such changes, namely, the fall-through block (i.e., the one starting at the instruction next to the currently executing one in the current code layout) or a different basic block.

**Instrumentation and Analysis Code** First of all, we check that the encountered instruction is a branch or a call instruction – thus a change in the flow of the program. Once we filtered only the instructions of interest, we want Pin to pass to our analysis callback the two information we pointed out previously:

```
void Ins(INS ins, void* v) {
    ...
    ADDRINT ins_end = INS_Address(ins) + INS_Size(ins);
    INS_InsertCall(ins, IPOINT_BEFORE,
        (AFUNPTR)INS_JumpAnalysis,
        IARG_ADDRINT,
        ins_end,
        IARG_BRANCH_TARGET_ADDR,
        IARG_BRANCH_TAKEN,
        IARG_THREAD_ID,
        IARG_END);
    ...
}
```

- *IARG\_ADDRINT ins\_end*: the address of the end of the instrumented instructions. We will understand the reason of collecting the address of the end, and not of the beginning of the instruction, at the end of Section 2.4.
- *IARG\_BRANCH\_TARGET\_ADDR*: the address of the next instruction that will be executed.
- *IARG\_BRANCH\_TAKEN*: whether the branch was taken. If this is true, we have altered the instruction pointer, while if it is false, the next instruction is the fall-through.
- *IARG\_THREAD\_ID*: the ID of the thread that is executing the instruction. We will explain in Section 2.2 why we need this.



In the analysis function we just assemble all the information we are given in a single string:

```
void INS_JumpAnalysis(ADDRINT ins_end,
                     ADDRINT target_branch,
                     INT32 taken,
                     THREADID thread_idx) {
    if (!taken) return;
    trace_t* trace =
        (trace_t*) PIN_GetThreadData(tls_key, thread_idx);
    /* Allocate enough space in order to save:
     - @ char (1 byte)
     - address in hex format
       (sizeof(ADDRINT) * 2) * 2 bytes for ip and target
     - \n delimiter (1 byte)
     - 0 terminator (1 byte)*/
    size_t buf_len = (sizeof(ADDRINT) * 4 + 3);
    // Trace limit guard
    if (traceLimitGuard(trace, buf_len, thread_idx)) return;

    char* buf = (char*) malloc(sizeof(char) * buf_len);
    MALLOC_ERROR_HANDLER(buf, "[x] Not enough space to allocate the
        buf for the INS_JumpAnalysis\n");
    sprintf(buf, "\n%08x%08x\0", ins_end, target_branch);
    ...
    /* Since this buf is either flushed or copied in memory we can
       free it */
    free(buf);
}
```

As we will see, depending on the different approach selected the function can either record the trace in memory, or flush the trace to the disk right away.

## 2.2 Multithreaded Programs

In multithreaded programs, we trace instructions in presence of concurrency, and thus we have to deal with race conditions and deadlocks.

Our approach is to use per-thread structures in order to avoid slowdowns introduced by the synchronization between threads. Nonetheless, there will still be race conditions during the creation of these structures. To deal with this we can use a simple lock provided by Pin to manage the access to critical sections. In our case, the critical section is represented by the creation of these per-thread structures.

The right time to initialize such structures is at a thread-start time, and fortunately, we are given the possibility to specify a callback that gets fired when a thread is started, and with it, we can set up all we need.

```
void ThreadStart(THREADID thread_idx, CONTEXT* ctx, INT32 flags, VOID* v) {
    PIN_GetLock(&pin_lock, thread_idx);
```

```

    /* Beginning of the critical section */
    char filename[TRACE_NAME_LENGTH_LIMIT] = { 0 };
    sprintf(filename, "trace_%d.out", thread_idx);
    FILE* out = fopen(filename, "w+");
    /* End of the critical section */
    PIN_ReleaseLock(&pin_lock);
}

```

To retrieve the per-thread structure we need to perform a lookup operation to find the structure linked to the current thread. To accomplish this in an efficient way, we use the Thread Local Storage (TLS) facility offered by Pin. With this, we have access to a dedicated space for each thread instrumented by the application. To access this storage it is sufficient to know the *thread\_id* of the current thread, and if we recall from Section 2.1, we make Pin pass to our callback an argument called *IARG\_THREAD\_ID*, which is exactly the *thread\_id* mentioned earlier that we need to access the TLS. Accessing the storage is as simple as in the snippet below:

```

void INS_Analysis(... , THREADID thread_idx) {
    trace_t* trace = (trace_t*)PIN_GetThreadData(tls_key, thread_idx);
    ...
}

```

We now have defined how we create our structures and how we access them without incurring in any sort of race condition or deadlock.

## 2.3 Challenges in the Analysis

While implementing our tracer, we have to keep in mind two major problems that we have to cope with:

- A significant **slowdown** of the application introduced by the analysis, which could be leveraged by some Malware to alter its execution and act benevolently.
- The **availability** of the code, that in case the program is packed is revealed only at runtime. We should thus provide means to store such code sections as they are revealed to allow for a later offline analysis.

**Slowdown.** To meet our performance needs we will experiment with different approaches to record our trace:

- **Unbuffered** version: the trace is flushed as soon as it is collected.
- **Buffered** version: the trace is stored in memory, and we flush the trace once we fill up the buffer or the instrumentation is about to end.
- **Thread Flushed** version: a dedicated thread is in charge of flushing the trace, without pausing the execution when we run out of available space in the buffer.

We will further discuss the implementation details in Chapter 3, and we will compare the different approaches in Chapter 4.

**Code Availability.** As far as the availability is concerned we will implement two techniques to deal with it:

- **Dump on exit:** we will dump the different sections of the program when it is about to finish, assuming that they will still contain the unpacked code.
- **$W \oplus X$  rule:** this rule imposes that a region of memory can either be written or executed, but not both. If a certain region of memory does not respect this rule, it probably means that code has been unpacked and then executed in that memory region [10]. We dump every region at each such violation.

## 2.4 CFG Reconstruction

The reconstruction of the control flow graph of the program will both help us in the evaluation phase, and gives us a graphical feedback of what we achieved. To reconstruct the CFG we will follow these steps:

- Parse the generated trace.
- Retrieve the disassembled code starting from recorded control flow decisions.
- Generate the CFG, and overlap it with the disassembled code.

First of all, we load a report generated by our Pintool with some relevant information to overlap the trace with the disassembled code

```
def load_report():
    ...
    images = report["images"]
    main_image = report["main_image"]
    sections = report["sections"]
    text_section = report['text_section']
    text_low = sections[text_section]['address']
    text_size = sections[text_section]['size']
    text_low = text_low
    text_size = text_size
    text_high = text_low + text_size
```

Then, we disassemble the code of the dumped executable section with the popular Capstone [1] code disassembly framework:

```
def disasm_text_section():
    ...
    instructions = []
    with open('.text.dump', 'rb') as f:
        for i in md.disasm(f.read(), text_low):
            instructions += [Instruction(address=(i.address), \\
                                         disasm=i.mnemonic + ' ' + i.op_str)]
    return instructions
```

Finally, we parse our trace, that has a structure like this:

FROM\_ADDRESS@TO\_ADDRESS

Once we have the disassembled code, we overlap it with our trace to generate our CFG with the Graphviz [4] visualization framework:

```
def getDisasmInRange(a:int, b:int):
    instructions = []
    for i in text_instr:
        if a <= i.address < b:
            instructions += [i.disasm]
    return instructions

def insertExternalStub(last_ip:str):
    found = False
    for name, mem_range in images.items():
        if mem_range['low_address'] <= int(last_ip, 16) \\\
            <= mem_range['high_address']:
            found = True
            short_name = name[name.rfind('\\')+1:]
            dot.node(last_ip, label=short_name, shape='ellipse')
    if not found:
        dot.node(last_ip, label='Stub', shape='ellipse')

def parse_trace():
    with open('trace_0.out') as f:
        lines_no = 0
        edges = set()
        last_ip = '0x0'
        for line in f:
            # Cleanup the string
            line = line.replace('\x00', '').strip()

            ip, target = line.split('@')
            # Very first instruction
            if ip == '':
                last_ip = target
                continue

            if not (text_low <= int(last_ip, 16) <= text_high):
                insertExternalStub(last_ip)
            else:
                instr_in_range = getDisasmInRange(int(last_ip, 16), \\\
                                                    int(ip, 16))
                dot.node(last_ip, label='\n'.join(instr_in_range))
                edges.add((last_ip, target))
                last_ip = target
```

```
        if lines_no >= TRACE_LIMIT:
            break
        lines_no += 1

    dot.edges(list(edges))
```

If the address of an instruction falls within the boundaries of the executable section of the main Image, we are able to retrieve its disassembled code, otherwise we insert a stub for the relative external DLL. The reason why we saved the address of the end of the instruction in 2.1 can be found in the *getDisasmInRange()* function. In fact, to correctly determine the range of the instructions in a block, we need the address of the end of the instruction.



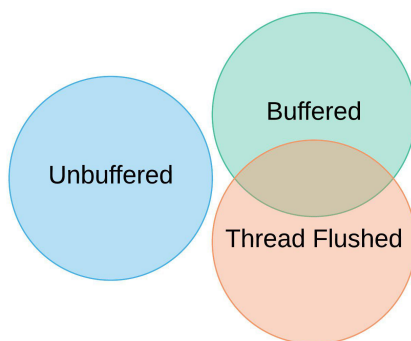
## Chapter 3

# Implementation

In this chapter, we will further discuss the three implementations proposed in Section 2.3. First, we will list some use cases for each of the proposed solutions:

- **Unbuffered:** This approach can be useful in a situation in which the execution of the program might damage the environment it is ran into<sup>1</sup>, and collecting the trace after the end of the program would be problematic. For example, we could run such a program in Virtual Machine, and record the pieces of the trace through a socket immediately after we collect it, in order to get as much information as possible.
- **Buffered:** This version is faster than the Unbuffered version and quite easy to implement. It would fit in any kind of scenario, apart from the one previously mentioned.
- **Thread Flushed:** The use cases of this technique are the same as the Buffered one, and although being more difficult to implement, it should perform faster.

The Buffered and the Thread Flushed versions share a lot of the logic. In fact, a Venn representation of the three solutions might be this one:



**Figure 3.1.** Venn diagram for use cases and features of the different versions.

Some code features are turned on or off in case we are in Buffered mode or not.

---

<sup>1</sup>In the malware analysis practice, malware samples sometimes retaliate against the analyst by destroying their machines when detected.

### 3.1 Unbuffered version

In this version, the trace we collect is immediately flushed and saved to the hard drive. There is also a trace guard, that in case the size of the trace grows too much, prevents other recordings from being carried out.

```
#define recordTraceToFile(f, buf, buf_len, trace) {\
    flushTraceToFile(f, buf, buf_len);\
    trace_size += buf_len;\
}

bool traceLimitGuard(trace_t* trace, size_t buf_len, THREADID thread_idx) {
    // If we reached the trace limit let's stop tracing
    if (trace_size + buf_len > trace_limit) {
        hasReachedTraceLimit[thread_idx] = true;
        return true;
    }
    // If we are in flushed mode no action is required
    if (!isBuffered) return false;
    ...
}

void INS_JumpAnalysis(..., THREADID thread_idx) {
    ...
    recordTraceToFile(files[thread_idx], buf, buf_len, trace);
    ...
}
```

As the snippet is straightforward, we believe no explanation is needed for the reader.

### 3.2 Buffered version

In this version, we opted for per-thread structures as we did with recorded trace files. This definitely speeds up the analysis, since there is no need to synchronize the different threads with each other. We aimed for simplicity in the implementation, keeping the overhead introduced by its usage the lowest possible.

Our trace structure is composed of: a buffer, in which the trace is stored; the size of the buffer, to check if the limit of the buffer size has been reached; and a cursor, to keep track of the last position we were writing at, allowing for  $O(1)$  writes.

There is also a `__pad` field, that stands for padding, and this is due to the fact that *false sharing* may occur. As the documentation reads: “False sharing occurs when multiple threads access different parts of the same cache line and at least one of them is a write. To maintain memory coherency, the computer must copy the memory from one CPU’s cache to another, even though data is not truly shared. False sharing can usually be avoided by padding critical data structures to the size of a cache line, or by rearranging the data layout of structures” [7].



We thus rearranged and padded our critical structure inserting the `__pad` field:

```
#define TRACE_PADDING CACHE_LINE_SIZE - 12
typedef struct trace_s {
    size_t cursor;
    char* buf;
    uint8_t _pad[TRACE_PADDING];
    size_t buf_size;
} trace_t;
```

The function in charge of enqueueing the logs into the buffers is pretty much the same as for the unbuffered version. Here the trace guard also checks the size of the buffer, and if we end up finishing the space, we dump the data collected. What we do is pausing the analysis, flushing the information, and then resuming tracing. To reduce the amount of time taken by this operation we do not release the dynamically allocated buffer; instead, we simply move the cursor back to the beginning.

```
#define recordTraceInMemory(buf, buf_len, trace) {\
    memcpy(trace->buf + trace->cursor, buf, buf_len);\
    trace->cursor += buf_len;\
    trace_size += buf_len;\
}

bool traceLimitGuard(trace_t* trace, size_t buf_len, THREADID thread_idx) {
    ...
    // If we have not reached the main buffer maximum size no action is required
    if (trace->cursor + buf_len <= trace->buf_size) return false;
    if (!isThreadFlushed) {
        time_t tv;
        INFO("[*] Thread buffer limit reached, flushing\n");
        if (!thread_idx) START_STOPWATCH(tv);
        flushTraceToFile(files[thread_idx], trace->buf, trace->cursor);
        if (!thread_idx) total_flushing_time += GET_STOPWATCH_LAP(tv);
        trace->cursor = 0;
    } else { ... }
    return false;
}

inline void INS_JumpAnalysis(...) {
    ...
    // Trace limit guard
    if (traceLimitGuard(trace, buf_len, thread_idx)) return;
    ...
    recordTraceInMemory(buf, buf_len, trace);
    ...
}
```

### 3.3 Thread buffered version

This version is the trickiest one. The real difference with the Buffered version lies in the way the trace is dumped when we fill up the space available for our trace buffer. Instead of letting the analysis thread doing the dump, we let another thread, running in parallel, do this job, and use another buffer for tracing. We need some synchronization between the different analysis threads and the flusher thread though.

Pin offers some *locking primitives* that can help us build our structure. The primitives we will use are of two types:

- *PIN\_SEMAPHORE*: a simple semaphore implementation that gives us the possibility to put some thread on hold for something to happen;
- *PIN\_MUTEX*: a simple lock to manage accesses to a certain critical section. Unlike *PIN\_LOCK* objects, here the concept of ownership is missing. We will see later why we need this characteristic.

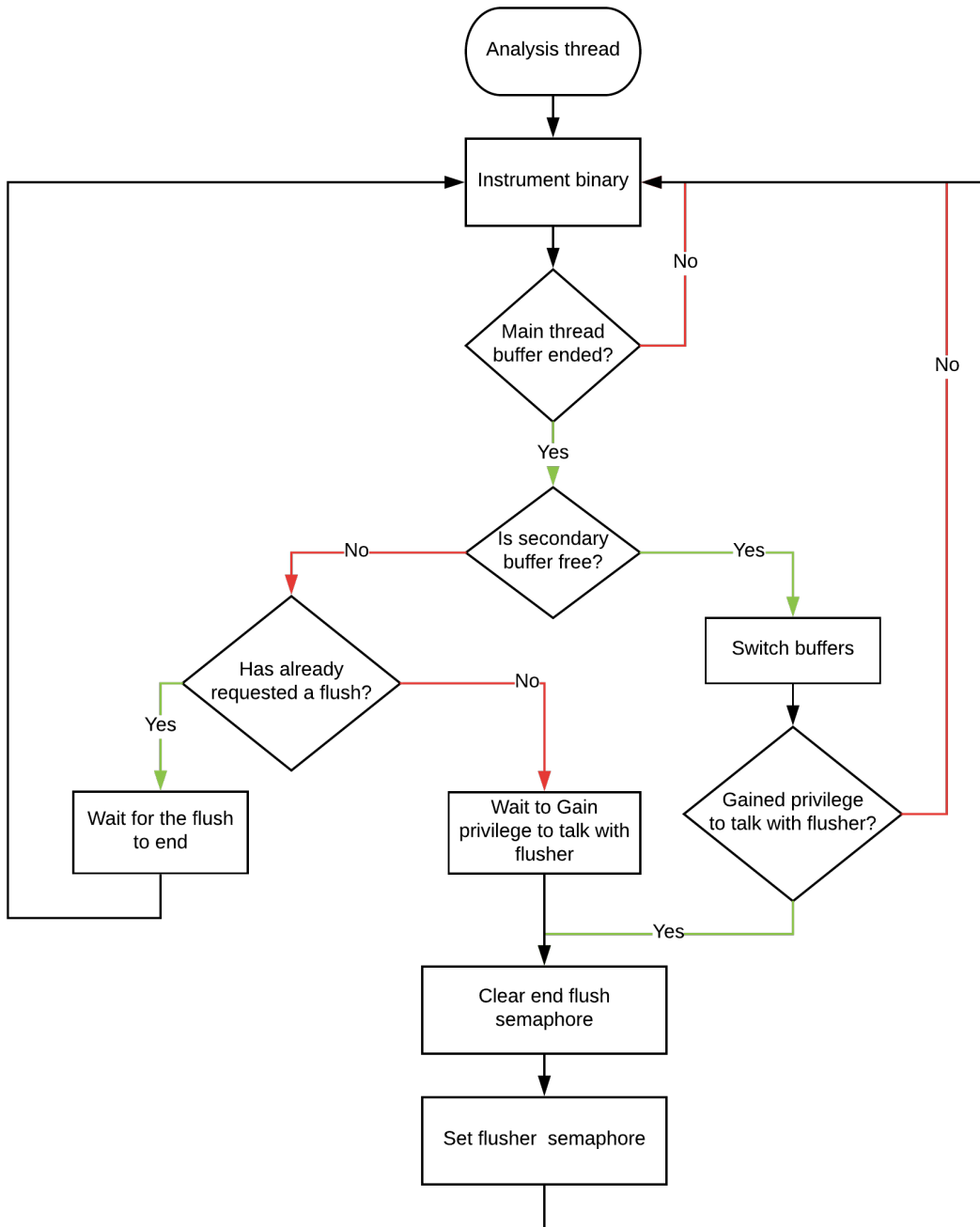
For starters, we need to create the flusher thread, and for doing within Pin we must use the given *PIN\_SpawnInternalThread(...)* function to spawn new “internal” threads. We do this in our *main(...)* function while setting the other callbacks up. Once the flusher thread is spawned, we now have to take care of the communication that occurs between it and the other analysis threads. The steps that each thread has to go through in order to complete a flush request are the following:

1. **Gain privilege** to talk with the flusher. Since the flusher is a singleton, only one thread at a time might be speaking with it.
2. Prepare all that is necessary for the flush.
3. Inform the flusher that it is the one who is to be served and that it should start flushing.
4. If needed, **wait** for the end of the flush.

As far as the flusher thread is concerned, what it needs to do is:

1. **Wait** for a thread’s request to undertake a flush.
2. Carry out the flush.
3. Inform the thread that requested the flush that the task ended.
4. Let other threads talk with it and go back to step 1.

We highlighted in bold the motivations for the introduction of the previously mentioned locking primitives. We used a *PIN\_MUTEX* to satisfy the “**Gain privilege**” request. Once obtained this privilege, an analysis thread sets everything up and lets the flusher do the rest. Had we used a *PIN\_LOCK* instead, this thread would have been in charge of releasing it, thus being forced to wait for the flusher to finish. This would have hindered the parallelism we are looking for to speed up the analysis. Since we used a *PIN\_MUTEX*, the flusher is the one who unlocks the mutex that regulates the possibility to talk with it.



**Figure 3.2.** The analysis threads' steps to interact with the flusher.

Although we try our best to avoid that an analysis thread has to wait for the flusher to finish, in case it fills up both its main and secondary buffer, it is forced to “**Wait** for the end of the flush”. To meet this criterion, we used a per-thread *PIN\_SEMAPHORE* that is: (i) cleared by the thread, while it prepares everything for the flush to happen; and is then (ii) set by the flusher, once it finishes its duty. The semaphore is per-thread because without having an owner of the *PIN\_MUTEX* to interact with the flusher, it might happen that a thread waits for the end of the flush of another thread.

This is the per-thread structure that we use in this version:

```
typedef struct doub_buf_trace_s {
    trace_t trace;
    bool isFlushBufEmpty;
    bool isFlushing;
    char* flush_buf;
    size_t flush_buf_len;
    PIN_SEMAPHORE end_flush_sem;
} doub_buf_trace_t;
```

We thus inherit all the fields of the original tracer. Then, we add the secondary buffer, the per-thread *PIN\_SEMAPHORE*, and some utility variables.

Last but not least, the flusher has to “**Wait** for a thread’s request”. We could have implemented this in a *busy waiting* fashion, but as we believe it would have been rather inefficient, we opted for using a *PIN\_SEMAPHORE* here as well. This semaphore is: (i) set by the thread asking for a flush; and then (ii) cleared by the flusher once it finishes flushing, to put itself in a kind of “sleep” state, waiting for another request to occur.

To better understand the conceptual flow described in this section, we provide in Figure 3.2 a flowchart that summarizes the various steps.

### 3.4 Dump the Code

When dealing with packed programs whose code is available only at runtime, we need to dump the code before finishing the analysis in order to allow subsequent offline analyses. As stated in Section 2.3, we can simply flush the different sections of a binary before exiting, or implement the  $W \oplus X$  rule to find out which memory ranges are to be dumped. For the first one, what we do is registering a callback that Pin will fire when the program is about to exit, and then, we loop through all the different sections of the program and dump them to an external file.

```
void dumpSections(IMG img) {
    for (SEC sec = IMG_SecHead(img);
        SEC_Valid(sec);
        sec = SEC_Next(sec)) {
        FILE* f = fopen((SEC_Name(sec) + ".dump").c_str(), "w+");
        char* sec_dump = (char*) malloc(SEC_Size(sec));
        PIN_SafeCopy(sec_dump,
                     (void*) SEC_Address(sec),
                     SEC_Size(sec));
        fclose(f);

        report_j["sections"][SEC_Name(sec)]["address"] =
            SEC_Address(sec);
        report_j["sections"][SEC_Name(sec)]["size"] = SEC_Size(sec);
    }
}
```

We discussed the need to save the address and the size of each section in Section 2.4. On the other hand, for the  $W \oplus X$  rule we need to register all the memory that is written by each executed x86 instruction and then check if the execution falls in one of these intervals. To speed things up, we create memory ranges as suggested in [10] (Figure 3.3). First, we add the analysis functions to record all the memory writes, and to control whether the  $W \oplus X$  rule is respected or not:

```
void INS_WriteAnalysis(ADDRINT at, ADDRINT size) {
    bool hasFoundRange = false;
    for each (pair<ADDRINT, ADDRINT> interval in
        written_mem_intervals) {
        // Check if low address is in range
        if (IN_RANGE(at, interval.first, interval.second)) {
            // If high address is bigger than
            // second interval expand it
            if ((at + size) > interval.second)
                interval.second = at + size;
            hasFoundRange = true;
        }

        // Check if high address is in range
        if (IN_RANGE(at + size, interval.first, interval.second)) {
            // If low address is smaller than
            // first interval expand it
            if (at < interval.first)
                interval.first = at;
            hasFoundRange = true;
        }
    }

    // If no range has been found, let's create a new one
    if (!hasFoundRange)
        written_mem_intervals.push_front(make_pair(at, at + size));
}

void INS_WXorX(ADDRINT at, const char* disasm_ins) {
    for each (pair<ADDRINT, ADDRINT> interval in
        written_mem_intervals) {
        if (IN_RANGE(at, interval.first, interval.second)) {
            if (upx_info->OEP == INVALID_ENTRY_POINT)
                upx_info->OEP = at;
            fprintf(upx_dump_file, "%s", disasm_ins);
        }
    }
}
```

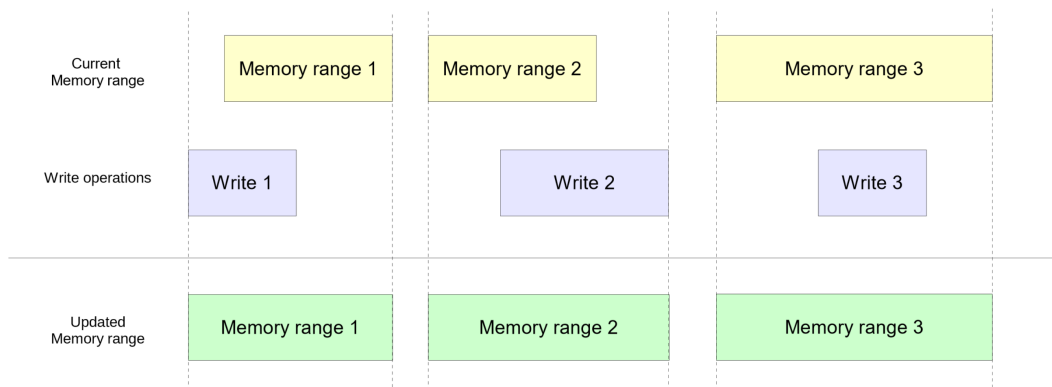
And then we register them:

```
void Ins(INS ins, void* v) {
    ...
    if (isBinaryPacked &&
        IN_RANGE(ins_addr,
                  main_img_memory.first,
                  main_img_memory.second)) {

        if (INS_IsMemoryWrite(ins)) {
            INS_InsertCall(ins, IPOINT_BEFORE,
                          (AFUNPTR) INS_WriteAnalysis,
                          IARG_MEMORYWRITE_EA,
                          IARG_MEMORYWRITE_SIZE,
                          IARG_END);
        }

        INS_InsertCall(ins, IPOINT_BEFORE,
                      (AFUNPTR) INS_WXorX,
                      IARG_ADDRINT,
                      ins_addr,
                      IARG_PTR,
                      disasm_ins,
                      IARG_END);
    }
}
```

Although the  $W \oplus X$  rules addresses also fancy features of complex packers such as dynamic allocation of code sections in heap<sup>2</sup>, early tests showed that the overhead introduced was enormous, hence we opted for the first strategy to carry out the evaluation phase of Chapter 4.



**Figure 3.3.** Memory ranges management

<sup>2</sup>This behavior is observed also in benign programs such as JavaScript browser engines.

# Chapter 4

## Evaluation

In this chapter, we will illustrate the methodology adopted to evaluate the outcome of our tests, the tools used to track the metrics we use, and the programs that we analyze.

### 4.1 Methodology

The methodology followed to evaluate our tool can be divided in: **performance assessment**, which is the measure of the impact the tool has on the execution of the original program; and **CFG reconstruction**, which is the measure of the quality of the information our tool gathered.

#### Performance assessment

We studied 3 different programs with our tool and measured **speed** (i.e., total time) of execution and **size** of the recorded trace. The programs analyzed are:

1. *FCIV*[3], a file integrity verifier run on the ISO image of Ubuntu 17.10 [5].
2. *driverquery* (with */V* option), a tool to display a list of installed device drivers and their properties.
3. *systeminfo*, a tool that displays detailed configuration information about a computer and its operating system.

We drew these programs from related research works. We will compare the total execution time of the original version with the Unbuffered, Buffered and Thread Flushed version. Then, we will compare some other statistics between the Buffered and Thread Flushed version to understand how effective the latter solution is.

For the *buffered* version we also measured:

- *Average time per flush* occurring every time a thread's buffer becomes full.
- *Flushing time*, i.e., time spent by analysis code flushing the trace.

And for the **thread flushed** version we measured:

- *Average time for each flush* that – as for the buffered version – occurs every time we finish the buffer space of a thread.
- *Flusher thread’s flushing time*, which is the time actually spent in flushing by the flusher thread.
- *Flusher thread’s running time*, which is the total amount of time the flusher was running. It includes the abovementioned *flusher thread’s flushing time*.

To gather this information we used:

- A *Powershell* script, to measure the duration of the whole instrumentation with the help of the *Measure-Command* `{...}` command.
- The `<time.h>` C library, to measure the duration of inner parts of the tool.

### CFG Reconstruction

In this part of the evaluation, we will compare the CFG of a program with the CFG of its packed version. First, we built the CFG of the observed programs. Then, we packed those programs with a well-known packer called UPX [9], we built the reconstructed the CFG of the packed version, and we compared the results.

To achieve a rigorous comparison we could have implemented a CFG similarity technique like the one based on graph edit distance mentioned in the well-known *A Generic Approach to Automatic Deobfuscation of Executable Code* work [6], but for our goals we obtain an almost perfect overlap since there are no external factors that could generate some sort of “noise” in the reconstruction process.



## 4.2 Results

We will now list the results obtained by the tests we carried out. The size of the trace will be the same for all the different techniques, so it will help us understand the overall impact of our tracer on the storage. On the other hand, time will help us declare which solution is better among the ones we implemented.

### Overhead

We first report the overhead introduced by each of the three techniques:

<b>Unbuffered</b>	Original (ms)	Instrumented (ms)	Trace size (Mb)	Overhead
systeminfo	2685	16815	45	6.3x
driverquery	600	60277	340	100x
fciv	7336	140237	920	19x

**Table 4.1.** Unbuffered version results

<b>Buffered</b>	Original (ms)	Instrumented (ms)	Trace Size (Mb)	Overhead
systeminfo	2685	12462	45	4.6x
driverquery	600	22697	340	37.8x
fciv	7336	36976	920	5x

**Table 4.2.** Buffered version results

<b>Thread Flushed</b>	Original (ms)	Instrumented (ms)	Trace Size (Mb)	Overhead
systeminfo	2685	12462	45	4.6x
driverquery	600	21543	340	35.9x
fciv	7336	34227	920	4.7x

**Table 4.3.** Thread Flushed version results

### Flusher quality

We then try to better investigate how much the flusher reduced the overhead introduced by the flushing time required for buffered strategies:

	Flushing time (ms)	Delta (ms)	Flusher overhead (ms)	Reduction
systeminfo	260	257	3	98.8%
driverquery	1200	1154	176	96.2%
fciv	2769	2749	241	99.3%

The delta between the Buffered and the Thread Flushed version derives from the saved flushing time. We can see from the tests that, with our solution, we managed to hide almost the entirety of the flushing time.

## CFG Reconstruction

For the sake of simplicity, we will only show the results for FCIV and its packed counterpart that we analyze and reconstruct.

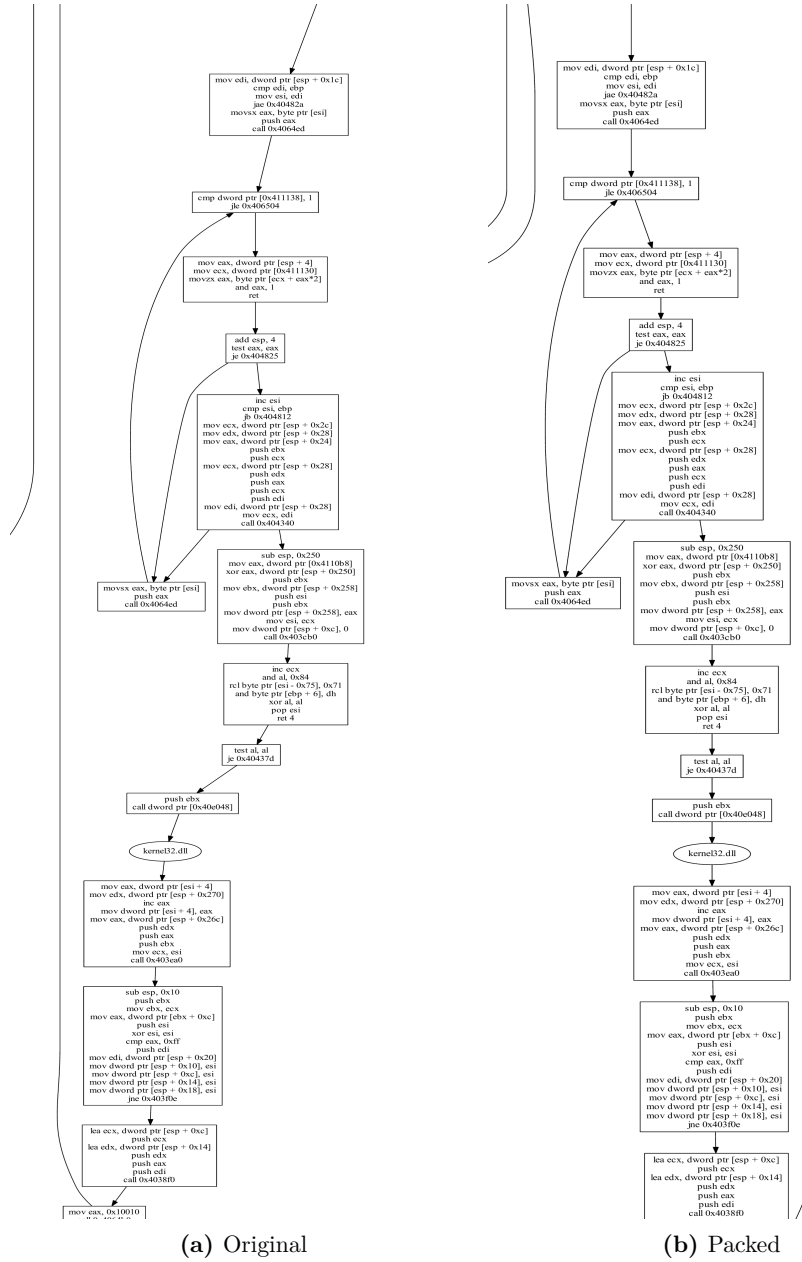
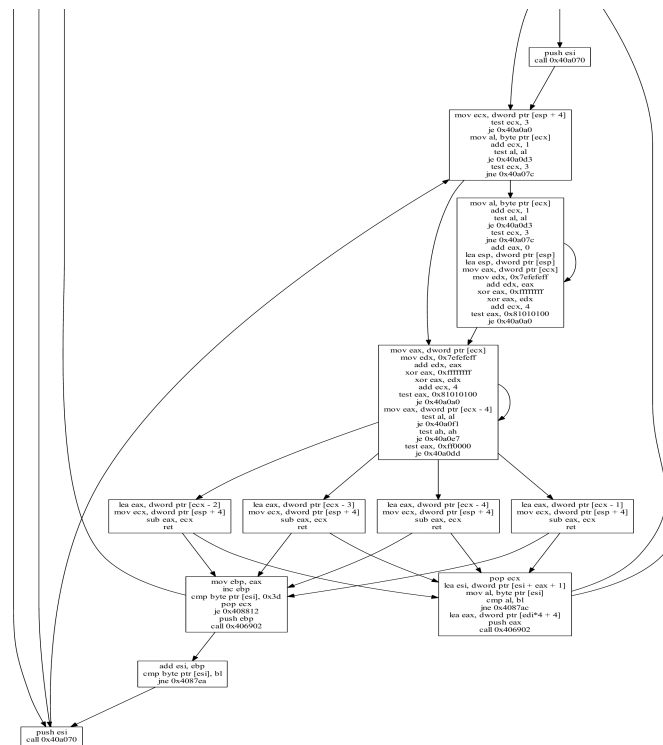
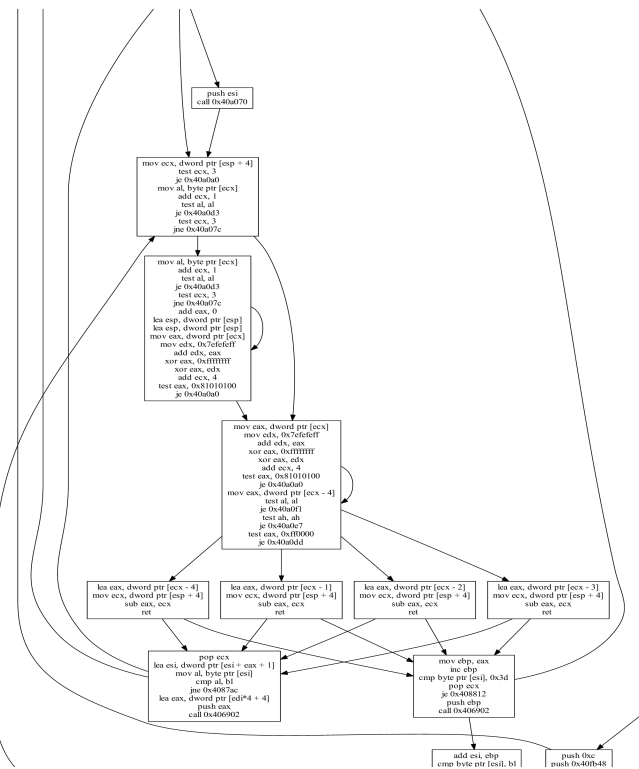


Figure 4.1. CFG (top part) of the original and packed version.



**Figure 4.2.** CFG (bottom part) of the original version.



**Figure 4.3.** CFG (bottom part) of the packed version.



## Chapter 5

# Conclusions

In this thesis, we have discussed different trade-offs that are possible in the design of an instruction tracer, in particular for malware analysis and reverse engineering applications of it. We started by introducing Dynamic Binary Instrumentation in Chapter 1, talking about the *modus operandi* of tools that adopt this technique. In Chapter 2 we outlined the elements needed to reach our goal of reconstructing control flow information of a program, and explained how and why we collected this information. Later on, we highlighted the challenges that building such a tracing tool introduces, and in Chapter 3 we discussed the implementations that we did for different proposals to cope with these challenges. Finally, we performed a preliminary evaluation of our solution in Chapter 4, showing that the engineering work behind the design of the solutions we proposed paid off in terms of improvements in the analysis performance. Also, an inspection of the graphs produced by our tool shows that it is able to correctly build the CFG of both a normal and a packed version of a simple yet realistic program.

Although the results are good, the targeted programs are fairly simple to handle, and I strongly believe that the presented solutions could be adapted to cope with more complex packers and obfuscations as well, including for instance those that make use of Self-Modifying Code or dynamic allocation of code sections on heap.



# Bibliography

- [1] Capstone: a lightweight multi-platform, multi-architecture disassembly framework. Available from: <http://www.capstone-engine.org>.
- [2] Cfg excerpt from a malware analysis session. Available from: [https://www.tophertimzen.com/resources/cs407/slides/week04\\_01-MalwareAnalysis.html](https://www.tophertimzen.com/resources/cs407/slides/week04_01-MalwareAnalysis.html).
- [3] Fciv: Microsoft file checksum integrity verifier. Available from: <https://www.microsoft.com/en-us/download/details.aspx?id=11533>.
- [4] Graphviz: Graph visualization software. Available from: <https://www.graphviz.org/about>.
- [5] Ubuntu 17.10 (artful aardvark). Available from: <http://releases.ubuntu.com/17.10>.
- [6] BABAK, Y., BRIAN, J., BENJAMIN, W., AND SAUMYA, D. A generic approach to automatic deobfuscation of executable code. (2015).
- [7] INTEL. Pin 3.5 user guide. Available from: <https://software.intel.com/sites/landingpage/pintool/docs/97503/Pin/html>.
- [8] KIM, H., DAVID, K., DAN, C., AND VIJAY, J. R. Using pin for compiler and computer architecture research and education. (2007).
- [9] MARKUS, O., LÁSZLÓ, M., AND JOHN, F. R. Upx: the ultimate packer for executables. Available from: <https://upx.github.io/>.
- [10] POLINO, M., CONTINELLA, A., MARIANI, S., D’ALESSIO, S., FONTATA, L., GRITTI, F., AND ZANERO, S. Measuring and Defeating Anti-Instrumentation-Equipped Malware. In *Proceedings of the Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA)*.