Hardware Accelerated Control-Flow Reconstruction in JIT Environments

Konstantinos Kleftogiorgos

Thesis submitted in partial fulfillment of the requirements for the

Master of Science degree in Computer Science

University of Crete
School of Sciences and Engineering
Computer Science Department
University Campus, Voutes, Heraklion, GR-70013, Greece

Thesis Advisors:
Prof. Evangelos Markatos,
Dr. Sotiris Ioannidis

UNIVERSITY OF CRETE
COMPUTER SCIENCE DEPARTMENT

JIT Tracing using Intel Processor Trace

Thesis submitted by
Konstantinos Kleftogiorgos
in partial fulfillment of the requirements for the
Master of Science degree in Computer Science

THESIS APPROVAL

Author: ______________________
Konstantinos Kleftogiorgos

Committee approvals: ______________________
Evangelos Markatos
Professor, Thesis Supervisor

____
Sotiris Ioannidis
Research Director, Committee Member

____
Maria Papadopouli
Professor, Committee Member

Departmental approval: ______________________
Antonis Argyros
Professor, Director of Graduate Studies

Heraklion, July 6th 2018

Abstract

Control-Flow reconstruction is a critical part of many security, profiling and analysis mechanisms. A challenging limitation in previous works is that they do not support tracing in JIT environments. Already existing mechanisms of obtaining the Control-Flow of a process, include the use of instrumentation, either dynamic or static. However, these approaches suffer from certain drawbacks. Obtaining the Control-Flow through dynamic instrumentation during the execution of a process, imposes severe slowdowns, while static instrumentation can lead to inaccurate results.

We leverage Intel Processor Trace, a new hardware feature of modern Intel CPUs, in order to acquire the Control-Flow of a process correctly, while at the same time minimizing the impact on the performance. Previous works have shown the effectiveness of utilizing Intel PT in order to reconstruct the Control-Flow of a process. However, none of them, to the best of our knowledge, has attempted to perform Control-Flow reconstruction on a process executing inside a JIT environment.

To showcase our mechanism in JIT environments, we trace the execution of a process in Intel Pin dynamic instrumentation framework. To achieve this we implemented a custom Intel PT driver and a new decoder which enables us to reconstruct the Control-Flow at runtime and not after the completion of the process. This approach imposes significantly less overhead, compared to dynamic binary instrumentation, while being more accurate than the static one. Finally, we evaluate the correctness of our mechanism and measure its performance by running SPEC2006 benchmark suit. Our results indicate that the overhead imposed by our mechanism, is marginally lower than previously developed mechanisms, while the Control-Flow is accurately reconstructed.

Περίληψη

Η ανακατασκευή της Ροής-Εκτέλεσης είναι ένα σημαντικό κομμάτι πολλών μηχανισμών ασφάλειας, δημιουργίας προφίλ και ανάλυσης. Ένας αξιοσημείωτος περιορισμός σε προηγούμενες δουλειές, είναι ότι δεν υποστηρίζουν την παρακολούθηση προγράμματος σε περιβάλλοντα JIT. Ήδη υπάρχοντες μηχανισμοί, για την απόκτηση της Ροής-Εκτέλεσης μιας διαδικασίας, περιλαμβάνουν τη χρήση παραμετροποίησης του πηγαίου κώδικα, είτε δυναμικά είτε στατικά. Ωστόσο, αυτές οι προσεγγίσεις υποφέρουν από ορισμένα μειονεκτήματα. Η λήψη της ροής ελέγχου μέσω δυναμικής παραμετροποίησης κατά την εκτέλεση μιας διαδικασίας, προκαλεί σοβαρή επιβράδυνση, ενώ η στατική μπορεί να οδηγήσει σε ανακριβή αποτελέσματα.

Χρησιμοποιούμε την τεχνολογία Intel Processor Trace, μια νέα δυνατότητα υλοποιημένη στο υλικό των σύγχρονων επεξεργαστών της Intel, προκειμένου να αποκτηθεί η Ροή-Ελέγχου μιας διαδικασίας σωστά, ελαχιστοποιώντας τις επιπτώσεις στην απόδοση. Προηγούμενες δουλειές έχουν δείξει την αποτελεσματικότητα της αξιοποίησης του Intel PT για την ανασυγκρότηση της Ροής-Εκτέλεσης μιας διαδικασίας. Ωστόσο, κανένας από αυτούς, εξ όσων γνωρίζουμε, δεν έχει προσπαθήσει να εκτελέσει ανασυγκρότηση της Ροής-Εκτέλεσης σε μια διαδικασία που εκτελείται μέσα σε περιβάλλοντα JIT.

Για να δείξουμε τη λειτουργία του μηχανισμού μας, βρίσκουμε τη Ροή-Εκτέλεσης ενός προγράμματος που εκτελείται μέσα στον παραμετροποιητή κώδικα Intel Pin. Γιάυτό το λόγο, υλοποιήσαμε έναν οδηγό υλικού για το Intel PT καθώς και ένα νέο αποκωδικοποιητή, ο οποίος μας επιτρέπει να ανασυγκροτήσαμε της Ροή-Εκτέλεσης κατά τη διάρκεια της εκτέλεσης και όχι μετά το τέλος του προγράμματος. Αυτή η προσέγγιση επιβάλλει σημαντικά λιγότερες καθυστερήσεις, σε σύγκριση με τη δυναμική παραμετροποίηση κώδικα, και είναι περισσότερο ακριβής από τη στατική. Τέλος, χρησιμοποιήσαμε τη σουίτα μέτρησης επιδόσεων SPEC CPU2006, προκειμένου να αξιολογήσουμε την αποτελεσματικότητά του πρωτοτύπου μας και να μετρήσουμε τις επιδόσεις του. Τα αποτελέσματα μας αποδεικνύουν ότι η επιβράδυνση του μηχανισμού μας είναι τάξης μεγέθους χαμηλότερη από τους προηγούμενους μηχανισμούς, καθώς παράλληλα επιτυγχάνει την ανακατασκευή της Ροής-Εκτέλεσης του προγράμματος.

Acknowledgments I would like to thank my Advisor Dr. Sotiris Ioannidis for his guidance and support during my studies and this work. I would also like to thank my Supervisor, Professor Evangelos Markatos, for his overall assistance and interesting discussions all these years. Many thanks to my Advisor during my internship at Stevens Institute of Technology, Dr George Portokalidis, and George Christou for his overall help on this project.

I need to also express my appreciativeness to Dimitris Deyannis, Hlias Papadopoulos, Eva Papadogiannaki, George Tsirantonakis, Rafail Tsirmpas, Christos Papachristos, Michalis Diamantaris, Panagiotis Ilia, Nick Christoulakis, Michalis Pachilakis, Dimitris Karnikis, Andreas Dedousis, Giannis Giakoumakis, Kostas Plelis, Manos Athanatos, Vaggelis Dimitriadis, Eirini Degleri, Giorgos Anagnopoulos, my colleagues and most important, friends in the Distributed Computing Systems (DCS) Lab. In like manner, I would like to thank my friends: Aggelos, Anna, Annastasios, Dimitra, Eirini, Giannis, Joanna, Kostas, Nikos, Palios, Pantelis, Panos, Rafail, Stauros. In addition, I would like to specially thank Monster Inc. and Myprotein.

Finally, I would like to sincerely thank my family, my father David, my mother Sofia and my sister Alexandra, for helping me all these years and supporting my choices. This work would have never been possible without them.

This work has been performed at the **Distributed Computing Systems laboratory, Institute of Computer Science, Foundation of Research and Technology – Hellas (FORTH)** in collaboration with the **Department of Computer Science, Stevens Institute of Technology**. In addition, this work has been supported in part by the European Commission via the H2020 ICT-32-2014 Project SHARCS under Grant 644571 and the H2020 DS-03-2015 Project CIPSEC under Grant 700378.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Various security defense mechanisms depend on correctly obtaining the Control-Flow of a process that they aim to protect. Acquiring the Control-Flow, while a process is executing through the existing techniques, imposes significant performance overhead. Moreover, less thorough approaches are inaccurate or not flexible. This is mainly because of the fact that, in order to acquire the exact Control-Flow that a process followed during its execution, instrumentation of the binary is required, either offline or at runtime. Incorrectly reconstructing the Control-Flow of a process, can lead to the false employment of several security, analysis and profiling mechanisms. For this reason, obtaining the Control-Flow of a process, in an accurate and stable way, regardless of the environment where it is executing, is quite critical and vital for several existing mechanisms. The obtained Control-Flow of a process can be utilized by several fields ranging from security [1, 2] and binary analysis [3, 4, 5], to profiling of applications [6, 7, 8, 9].

Until today, the existing techniques of obtaining the Control-Flow of an application suffer from several drawbacks, that include performance overheads and inaccuracies. We have developed a reliable and accurate hardware accelerated Control-Flow reconstruction system, that can trace the execution of a process with *marginally* less overhead than the existing mechanisms and independently of the environment where this process is executing. Our approach leverages Intel Processor Trace (PT) [], a new hardware feature that ships with the newest Intel CPUs. Intel PT was designed to aid in debugging and failure diagnosis by recording the minimal information necessary to reconstruct complete Control-Flow traces. We utilize Intel PT to reconstruct the Control-Flow of any process, regardless of the environment where it is executing, even if that environment contains JIT code. This is accomplished by isolating the execution flow of the original code from inside the JITed one.

The following sections present some of the appliances of the Control-Flow of a process.

1.1 Control-Flow Integrity

Control-Flow Integrity [10] (CFI) is a runtime security scheme that aims at guaranteeing that a given program will adhere to its intended Control-Flow Graph (CFG). CFI focuses on reinforcing indirect branches, like calls and returns, so that they become immune to tampering from memory corruption vulnerabilities like buffer overflows. CFI defenses are considered to be the most effective defenses against code-reuse attacks [11]. This kind of attacks aim to exploit compromised programs by subverting their Control-Flow, from the expected one.

Forward-Edge Control-Flow Integrity refers to the reinforcement of forward-edge branches, i.e. Indirect Call Instructions. Indirect calls utilise function pointers stored in memory to find their target. Unfortunately, these pointers are not safely stored in memory and are susceptible to corruption by tampering. Forward-Edge CFI validates that the indirect call reached a target that belongs to a group of authorised branch targets. This limits the call to extremely few indirect branch targets, hopefully none that the attacker could exploit.

Backward-Edge Control-Flow Integrity refers to the reinforcement of backward-edge branches, i.e. Return Instructions. Returns, like indirect calls, utilize a pointer stored in the stack as a branch target. This pointer is extremely vulnerable to tampering since it resides in the stack. Backward-Edge CFI keeps track of the return addresses and verifies that each return instruction did indeed reach its target.

However, attacks are still possible even after confining program execution to a CFG, through the use of CFI. Throughout the years, researchers have focused on software-only implementations of CFI. Implementing CFI on software requires being able to trace the indirect calls of the process during the execution. Several approach trying to achieve this exist, and can be divided into three categories:

Compile-Time Instrumentation
The main advantage of compile-time instrumentation [12, 13, 14] is that it has better performance than other approaches, since the instrumentation is performed offline. However, this approach suffers from several limitations. Such techniques can only instrument programs supported by the compiler and they require the source code of the application to be provided. In addition, the produced CFI, through compile-time instrumentation, is fixed for each application, forbidding the systems from customizing it at runtime.

Static Binary Instrumentation
Static binary instrumentation [10, 15] offers the ability to instrument programs written in a variety of languages and legacy binaries. However, the enforced CFGs are not as accurate as compile-time, while at the same time some compile-time optimizations cannot be applied. In addition, the produced instrumentation is fixed through the execution of the process.

Runtime Instrumentation
Runtime instrumentation [16, 17] offers the flexibility of instrumenting the process

during its runtime. This approach is more accurate than the fixed one, but it suffers from far greater overheads.

1.2 Dynamic data Flow Tracking

A notable technique proposed in the literature, in order to counter buffer-overflow related exploits, is Dynamic data Flow Tracking (DFT). The key concept of this mechanism, is to taint memory regions where untrusted data are residing, and track their propagation in the address space. Also, any new data resulting from computation with tainted data, becomes tainted as well. Exploits are detected with a predefined policy, depending on the implementation of DFT. In general, when tainted data are used in a suspicious, manner a security exception is raised. A common security exception trigger is when tainted data are used as an indirect jump operand. For example, in the case where an attacker overwrites a return address, by exploiting a buffer overflow, input data will be tainted and the DFT policy will detect the violation since the return address will also be tainted. In the majority of DFT implementations, the protected application is oblivious of the mechanism. Thus there is no need for source code modifications.

Dynamic data flow tracking is being used extensively in security research for protecting software [1, 2], analyzing malware [18, 19], discovering bugs [9, 20], reverse engineering [21], information flow control [22, 23], etc. However, dynamically applying DFT tends to significantly slow down the target application, especially when a virtualization framework [24, 25] is used to apply it on binary-only software. Overheads can range from a significant percentage over native execution to several orders of magnitude, depending on the framework used and particular traits of the implementation

1.3 Control-Flow Graph Generation

A Control-Flow graph (CFG) is a representation, using graph notation, of all paths that might be traversed through a program during its execution. A lot of works have focuses on generating an accurate CFG [26, 5, 27, 28]. However, most of them follow an offline approach that is based on static analysis of the binary. Profiling a binary dynamically, by monitoring the control-flow during the process's execution offers greater accuracy. An accurate Control-Flow Graph can enhance those techniques.

1.4 Contributions

All the previously mentioned reasons indicate that a correctly obtained Control-Flow of a process, can be quite useful to several research fields. Our work is based on this assumption and contributes the following.

- We utilize Intel PT, to trace the execution of a process and reconstruct its Control-Flow during runtime. This is performed accurately and faster than the previous methods that require the binary to be instrumented.

- We introduce a novel technique for tracing the execution of a process that is executed inside a JIT environment. Our Control-Flow reconstruction algorithm is able to isolate the execution of the JITed process and correctly reconstruct its Control-Flow.
- We implemented a prototype that is able to successfully reconstruct the Control-Flow of JITed applications at runtime. We showcase our mechanism by tracing the execution of a binary on Intel PIN binary instrumentation framework. We tested the correctness of our prototype by comparing it to the Control-Flow acquired using dynamic instrumentation.

Chapter 2

Background

In this chapter we provide a brief overview of the background concepts and technologies of this work. This background is essential in order to comprehend further chapters of this thesis. Chapter 2.1 describes the basic idea of Control-Flow and the concepts that constitute it. Chapter 2.2 introduces briefly the Intel IA-32 virtualization extension to provide virtualization by the hardware itself. In chapter, 2.3 the Intel Pin instrumentation framework overview is presented, together with some of its main APIs. Finally, the remaining chapters present other tools or features used throughout our implementations, such as Intel XED, POSIX Shared Memory and Loadable Kernel Modules.

2.1 Control-Flow

During a process's execution, the Control-Flow is the order in which individual statements and function calls of the process are executed. Within an executing process, a Control-Flow statement is a statement the execution of which, results in the decision made as which of the two paths to follow. Specifically, a `branch` is either taken or not taken. On the other hand, during a `call` or `jump` instruction the path is set as the target address. Since our system focuses on reconstructing the Control-Flow of a program executed inside a Just-in-Time environment, it seems appropriate to describe some of the main definitions that constitute it.

2.1.1 Branch Instructions

A branch is an instruction in a computer program that can cause a control-flow transition. The processor will start executing an instruction sequence not adjucent to the previous program counter. Thus, deviate from its default behavior of executing instructions in order. This behavior can be caused by the following:

Conditional Branches:
Conditional Branches consist of instructions that direct the execution of the program to another part of the program, only if a certain condition is true. If this specific

conditions is met, the control transfers to a new address and the execution goes on from this point onward.

Direct Calls:
Direct Branches refer to those instructions that cause the execution of the program to transfer to a specific address. This address is constant for a particular binary and the execution flow of this program will always follow this direction.

Indirect Branches:
Indirect Branches consist of jump or call instructions that, rather than directly specifying the address of the next instruction to be executed, they specify the location where the address resides i.e. the target address is calculated at runtime. The target address can either be stored in a register or a memory location.

2.1.2 Basic Blocks

A Basic Block is an instruction sequence with no branches targeting to it, except to the entry, and no branches targeting out, except at the exit. This means that, no code within this block is the destination of a jump or branch instruction anywhere in the program and only the last instruction can cause the program to begin executing code in a different Basic Block. Under these circumstances, whenever the first instruction in a Basic Block is executed, the rest of the instructions are necessarily executed exactly once and in order. Throughout this work, by Control-Flow reconstruction we refer to the extraction of the Basic Blocks sequence that the traced program followed during its execution. The complete procedure of acquiring these Basic Blocks will be described in detail in section 4.

2.2 Intel Processor Trace

Intel Processor Trace (PT) is a new hardware feature present on recent Intel Architectures. It was first supported in Intel Core M and 5th generation Intel Core processors that are based on the Intel micro-architecture code named Broadwell. Intel PT provides execution and branch tracing information of a process directly by the processor with negligible overhead. Unlike other branch tracing technologies such as Intel Last Branch Record (LBR), the size of the output Buffer is no longer strictly limited by hardware. The only limitation is the size of the output Buffer, that could be filled up after tracing a long process. However, as long as the output Buffer is consumed in an adequate manner, Intel PT can offer long-term tracing of a process while avoiding loss of trace data due to buffer overflows. In general, whenever a program is configured to use Intel PT, trace packets, encoding Control-Flow information, are generated throughout its entire execution. Intel PT packets are separated into two different types: general execution information and Control-Flow information packets. A software decoder is able to reconstruct the entire Control-Flow of the traced program, by combining the recorded packets with the executed binary.

2.2.1 Execution Information Packets

Intel PT produces a set of miscellaneous packets describing various events and states of the CPU during trace sessions. This includes the following packet types:

Packet Stream Boundary (PSB):
PSB packets indicate the boundary of an Intel PT trace sample and configure an Intel PT software decoder in order to discover the beginning of the actual trace data. PSBs are included automatically by the CPU in the Intel PT data stream and no option is provided to deactivate the generation of PSBs.

Time-Stamp Counter (TSC):
TSC timestamp packets are produced in order to provide timing information. More specifically, they store the value of the software accessible time-stamp counter of the associated CPU at the moment during record. The generation of this packet type can be deactivated by modifying the IA32_RTIT_CTL.TSCEn MSR field. This packet type was not essential for this work.

Core Bus Ratio (CBR):
CBR packet contains the value of the bus clock ratio. This packet type is not required in order for the Control-Flow to be safely reconstructed and thus it was not used during this work. Unfortunately, this packet type is not configurable and its generation cannot be deactivated. However, each occurrence of this packet in the trace data can be safely ignored.

Overflow (OVF):
As already mentioned, it is possible that the configured output buffer might be overrun during runtime. In such case, an OVF packet is generated by the CPU to indicate to the Intel PT software decoder loss of packets. OVF can occur for any generated packet, thus it can potentially indicate loss of useful packets i.e. flow information packets.

Paging Information Packet (PIP):
PIPs indicate the modification of the CR3 register during runtime. As a result, the Intel PT software decoder is able to comprehend and handle a paging transition.

2.2.2 Flow Information Packets

Intel PT also produces different types of Control-Flow related packets during runtime. A software decoder is able to reconstruct the Control-Flow of a program by decoding those packets and by using them as directions in the actual process binary text segment.

Taken-Not-Taken (TNT):
If the processor executes any type of conditional jump, the former decision whether this jump was taken or not will produce a TNT packet containing the decision of the conditional branch. Those packets are essential for the reconstruction of the exact

Control-Flow of the traced program. A single TNT packet tracks the direction of up to 6 conditional branches.

Target IP (TIP):
If the processor executes a jump or transfer instruction, which depends on a value in memory or register, the decoder will not be able to recover the Control-Flow. TIPs are the packets that are created when the process's execution reaches an indirect call, a return, an exception or an interrupt. Those TIP packets store the corresponding target IP that was executed by the processor after the transfer occurred. These packets contain the target IP, although that address value may be compressed by eliminating upper bytes that match the last IP. By default, return instructions do not cause the hardware to produce TIP packets, and thus the decoder is responsible to track the state of the current call stack. However, if a special CPU feature called RET compression is deactivated, the CPU will include TIP packets even if a return instructions is being executed. In order to minimize complexity, we disabled RET compression in this work. Moreover, it is essential to disable RET compression in order for our tool to be used as a Control-Flow Integrity monitor.

Flow Update Packets (FUP)
FUPs are basically hint packets and indicate asynchronous events such as any type of interrupts or traps. They are usually followed by a TIP packet containing the address of the following instruction.

Packet Generation Enable Packets (PGE)
PGEs provide the IP at which the tracing of the program begun.

Packet Generation Disable Packets (PGD)
PGDs indicate the last traced packet before tracing was eventually disabled.

MODE:
If a processor mode switch occurs (e.g. from Protected Mode to Long Mode), the processor also notifies such an event in the form of a MODE packet.

2.2.3 Intel PT Software Decoder

In order to reconstruct the Control-Flow using the output data produced by Intel PT, the flow information data must first be decoded. For this reason we employ Intel Processor Trace Decoder Library (libipt) [29], offered by Intel. libipt is a reference implementation for decoding Intel PT data and it can be used as a standalone library or it can be partially or fully integrated into a tool.

Intel PT, in order to reduce the amount of data produced by the processor, does not provide the actual executed instruction pointer but generates as little information as possible. For this reason the Intel PT software decoder does not only require the Control-Flow information data to reconstruct the Control-Flow, but also needs the program that was executed during tracing. Our work focuses on programs executed inside a JIT environment. In general, Just-in-Time compilers tend to modify the actual program code during runtime, making the reconstruction of the flow more complicated. In order to bypass this limitation, our decoder needs the actual executed JIT code, alongside with the original code of the executable file.

2.2.4 Trace Filtering

To limit the amount of generated trace data, Intel PT provides multiple options for runtime filtering:

CPL-Filtering:
Current Privilege Level Filtering offers the option to choose the privilege level that will be traced by Intel PT. Intel PT provides the ability to configure a logical processor to generate trace packets only when CPL = 0, when CPL > 0, or regardless of CPL. Out tool utilizes this filtering option to limit tracing explicitly to user mode execution, since Kernel space code is not required for reconstructing the Control-Flow.

CR3-Filtering:
The Intel PT hardware also provides CR3 filtering. CR3 filtering is based on address spaces. This means that the execution trace is only recorded if the CR3 register value matches the CR3 filter value. Since CR3 values can be considered as process identifiers, you use this feature to only record code executed in the context of a certain process. For the purpose of our tool, we use CR3 filtering in order to trace the execution of the JIT code.

IP-Filtering:
When tracing the target software execution, the Intel PT hardware records every event. To focus on information of your interest, you can use IP filtering. Using this feature, you can specify address ranges where the software execution is traced. Depending on the processor, it might be possible to configure multiple instruction pointer filter ranges. In general, those filter ranges only affect virtual addresses if paging is enabled. Our system utilizes this filtering option, in order to trace the execution of the Code Cache, where the generated code resides. The collected trace information are adequate for the Control-Flow reconstruction to be performed.

2.2.5 Table of Physical Addresses (ToPA)

Intel PT output should be viewed independently from trace content and filtering mechanisms. The options available for trace output can vary across processor generations and platforms. Trace output is written out using one of the following output schemes, as configured by the ToPA and FabricEn bitfields of IA32_RTIT_CTL.

- A single, contiguous region of physical address space.
- A collection of variable-sized regions of physical memory. These regions are linked together by tables of pointers to those regions, referred to as Table of Physical

Addresses (ToPA). The trace output bypasses the caches and the TLBs, but are not serializing. This is intended to minimize the performance impact of the output.

- A platform-specific trace transport subsystem.

This work utilizes only the ToPA mechanism. In general, The Table of Physical Addresses is a mechanism that allows multiple distributed memory chunks to be concatenated together to a unified contiguous output region. The ToPA mechanism uses a linked list of tables. Each entry in the table contains some attribute bits, a pointer to an output region, and the size of the region. The last entry in the table may hold a pointer to the next table. This pointer can either point to the top of the current table, in order to create a circular array, or to the base of another table. The table size is not fixed, since the link to the next table can exist at any entry. The processor treats the various output regions referenced by the ToPA tables as a unified Buffer. The ToPA mechanism is controlled by three values maintained by the processor:

proc_trace_table_base
This is the physical address of the base of the current ToPA table. When tracing is enabled, the processor loads this value from the IA32_RTIT_OUTPUT_BASE MSR.

proc_trace_table_offset
This indicates the entry of the table that is currently in use, meaning that it contains the address of the current output region. When tracing is enabled, the processor loads this value from bits 31:7 of the IA32_RTIT_OUTPUT_MASK_PTRS.

proc_trace_output_offset
This is a pointer into the current output region and indicates the location of the next write. When tracing is enabled, the processor loads this value from bits 63:32 of the IA32_RTIT_OUTPUT_MASK_PTRS.

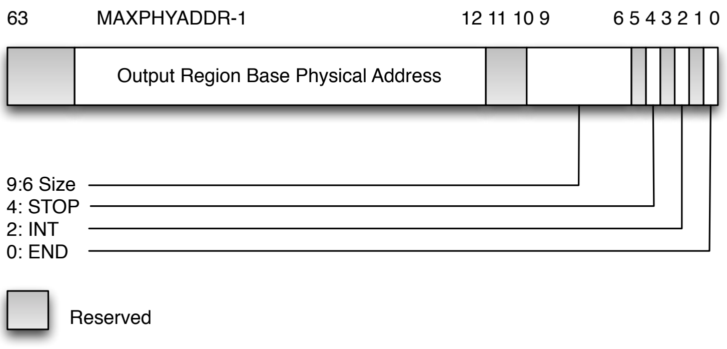


Figure 2.1: ToPA Entry Overview.

The format of ToPA table entries is shown in Figure 2.1 Each ToPA entry is specifically encoded and contains a physical address, a size specifier for the referred memory chunk in physical memory and multiple type bits. These type bits designate a specific behavior on access of the ToPA Entry. The individual type bits consist of the following:

STOP Entry:
If the linked output region is filled, tracing will be disabled by hardware. The last ToPA entry must configure this option.

INT Entry:
If the associated ToPA output region is filled, a notifying performance-monitoring interrupt (PMI) will be raised and the trace generation will be continued in the next ToPA output region.

END Entry:
This entry is necessary for the ToPA structure to indicate the last entry within this table. Therefore, this ToPA entry points to the next ToPA table instead of a ToPA output region. By configuring this entry to point to the first entry of the ToPA table, a Ring Buffer is created. Our work employs this setup, as described in the following sections.

2.3 Intel Pin

Pin [30] 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. The tools created using Pin, called Pintools, can be used to perform program analysis on user space applications on Linux, Windows and OS X*. Pin offers run time instrumentation on already compiled binary files, thus, it requires no recompilation of the source code while supporting instrumentation of programs that dynamically generate code. In general, a Pintool registers instrumentation callback routines with Pin that are called whenever code instrumentation needs to be generated. These callback routines represents the instrumentation component. For our needs, we have utilized Pin and its API, in order to dynamically instrument our application and collect the required information for the next steps of our implementation. Our tool's main objective is to track a program's execution and by collecting certain information about Control-Flow decisions, to reconstruct its Control-Flow. Intel PT is able to collect information about indirect jumps, calls and conditional branches, however certain information need to be collected by instrumentation of the binary at run time.

In general, Pin consists of a virtual machine (VM), a Code Cache, and an Instrumentation API invoked by Pintools. After Pin attains control of the application, the VM organizes its components in order to execute it.

2.3.1 Just-in-Time Compiler

The best way to think about Pin is as a native to native "Just in Time" (JIT) compiler. The input to this compiler is the bytecode of a regular executable. Pin intercepts the execution of the first instruction of the executable and generates new code for the instruction sequence starting at this instruction. This generated code is then placed inside a specified memory region called the *Code Cache*. In order for the generated code to be executed the control transfers to the Code Cache. The generated code sequence, called Trace, is almost identical to the original one. Traces usually begin at the target of a taken branch and end with an unconditional branch, including calls and returns. Pin guarantees that a trace is only entered at the top, even though it may contain multiple exits. At the same time it ensures that it regains control when a branch exits the Trace. After regaining control, Pin generates more code for the branch target and continues execution. Pin makes this efficient by keeping all of the generated code inside the Code Cache so it can be reused and directly branching from one sequence to another. In JIT mode, the only code ever executed is the generated code. The original code is only used for reference. When generating code, Pin gives the user an opportunity to inject their own code through the instrumentation API.

For optimization reasons, while the instrumented Traces are executed, links between them are created, based on the Basic Blocks they contain. This optimization is performed by Pin, in order to reduce execution transitions to the Code Cache. By linking Traces together, a single access to the Code Cache will result in executing not only the Trace residing at the address that the Code Cache was entered, but all the linked Traces as well. All the above indicate that, the only way in order to track the execution of the original program using Intel PT, is by collecting information about the execution of the entire Code Cache.

2.3.2 Instrumentation

Pin allows a tool to insert arbitrary code written in C or C++ in specific places in the executable. The code is added dynamically while the executable is running. Pin provides a rich API that abstracts away the underlying instruction set . The Pintool is able to utilize this API in order to register instrumentation callback routines with Pin that are called whenever new code needs to be generated. This instrumentation callback routines represent the instrumentation component. It inspects the code to be generated, investigates its static properties, and decides if and where to inject calls to analysis or instrumentation functions. The Pintool can also register notification callback routines for events such as thread creation, forking, linking between Traces or Code Cache execution accesses. These callbacks are used in our tool in order to gather data, initialize/update certain data structures and inspect when the execution transfers to the Code Cache.

As previously mentioned, Pin's instrumentation is "Just in Time". Instrumentation occurs immediately before a code sequence is executed for the first time. We call this mode of operation, Trace Instrumentation. Trace instrumentation utilizes the Pin Trace API. Trace instrumentation lets the Pintool inspect and instrument an executable one trace at a time. Pin breaks the trace into basic blocks, BBLs. A BBL is a single entrance, single exit sequence of instructions. Using the Pin BBL API it is possible to insert an analysis call or a callback routine for each BBL or any instruction inside a specific BBL.

Throughout this work, Pin is used as the JIT environment of our preference in extracting and reconstructing the Control-Flow of the original program. We employ Trace, BBL and Instruction API in order to collect useful information that will assist us during the reconstruction of the Control-Flow.

2.4 Intel XED

As previously mentioned, in order for the program executed inside the JIT to be reconstructed the exact executed instructions of the Code Cache are required. We achieve this by obtaining every instruction using X86 Encoder Decoder (XED) [31]. XED is a software library for encoding and decoding X86, IA32 and Intel64, instructions. The decoder takes sequences of up to 15 bytes along with machine mode information and produces a data structure describing the opcode, operands, and flags. Disassembly is essentially a printing pass on the data structure. We utilize XED in order to disassemble every instruction of a Trace after being placed inside the Code Cache.

2.5 POSIX Shared Memory

POSIX Shared Memory is an inter-process communication (IPC) mechanism defined in POSIX specification. After setting up the shared memory, two or more processes may read from and write to the shared memory region. Compared to other IPC mechanisms (e.g. pipe, socket, etc), POSIX shared memory does not impose copy overheads, thus it can be quite useful in some cases. We utilize POSIX Shared Memory throughout our work in order to transfer JIT code information, required for the reconstruction, between Pin and the Intel PT Decoder.

2.6 Loadable Kernel Module

A loadable kernel module (LKM) is a mechanism for adding code to, or removing code from, the Linux kernel at run time. They are ideal for device drivers, enabling the kernel to communicate with the hardware without it having to know how the hardware works. The alternative to LKMs would be to build the code for each and every driver into the Linux kernel, which is not efficient as it would make the Linux kernel huge by having it support every hardware by default. Kernel modules are essentially pieces of code that can be loaded and unloaded into the kernel, upon demand, and extend the functionality of the kernel without the need to reboot the system.

In our work, we extended Simple-PT [32], a simple kernel driver for Intel PT offered by Intel, based on our needs. Simple-PT enables the collection of Intel PT traces for debugging reasons, and thus quite a few changes were required. All the modifications will be discussed in section 3.

2.6.1 Kernel Module Parameters

Module parameters give the ability to the user to pass a value of some kind to the Kernel Module upon loading, and also examine its value while the module is loaded and running.

In addition these parameters can be modified on the fly if required. The module_param() macro is used in order to pass a value to a Module parameter, while module_param_cb() allows run-time changes to be applied. This functionality is used by our Driver in order to allow a user, by configuring some Module parameters, to control the Intel PT hardware.

2.6.2 Input/ Output Control

Input/ Output Control, or IOCTL, is a system call for device-specific input/ output, that solves the problem of communicating from user space with the Device Driver. This mechanism provides the user with a generic call which can be used to control any device. By using IOCTL the user is able to write to the device as well as read data from it. Our Driver utilizes the IOCTL mechanism to expose certain functionality to the User space, that will be used throughout our system's implementation.

2.6.3 Blocking I/O

Whenever a process has to wait for an I/O event its execution is suspended. Suspended processes free the processor for other uses. There are several ways of handling sleeping and waking up processes in Linux, each suited to different needs. Most of those are based on the wait queue data structure. The wait queue stores all processes that are suspended till a specific event occurs. Wait queues are declared and initialized using the DECLARE_WAIT_QUEUE_HEAD macro. Once the wait queue is declared and initialized, a process may use it to go to sleep. We implemented sleeping by calling wait_event_interruptible() to the wait queue. This macro is the preferred way to sleep on an event. It combines waiting for an event and testing for its arrival in a way that avoids race conditions. It will sleep until the condition, which may be any boolean C expression, evaluates true. The macro expands to a while loop, and the condition is reevaluated. The wake_up_interruptible() macro wakes the process if the corresponding condition is met.

Chapter 3

Design

Tracing the execution of a process inside a JIT environment is not straightforward. One of the most important problems is that the JIT engine, in order to instrument the binary, requires to change the Basic Blocks of the original binary and create its own sequence of code. This new sequence of code is then placed inside a Code Cache, as Traces of code, and is executed by transferring the Control of the execution there. The goal of this work is to differentiate and extract the Basic Blocks of the original program from those produced by the Pintool and be able to reconstruct the complete Control-Flow of the instrumented binary. In order to achieve this, we have build our system by utilizing the Intel Processor Trace hardware feature, that comes with the newest Intel Processors. This section offers an overview of the complete Control-Flow reconstruction system and its individual components.

3.1 Architecture

3.1.1 Overview

Figure 3.1 presents the Overview of our Architecture. The main idea of our work is to create a system that is able to trace the execution of a program inside a JIT environment and reconstruct its Control-Flow. Our system uses a Pintool in order to spawn the main process. Pintool is the JIT environment that we are using throughout our implementation. In order for Intel PT to be able to collect trace data for Pintool, some actions need to be performed. Our Driver takes this responsibility by setting all the required registers, in order for the Pintool to be traced, as well as starting Intel PT. Intel PT is now able to begin tracing of the Pintool. All the produced data are written in the Ring Buffer, that consists of several sub-buffers. The Decoder is blocked by the Driver and is waiting for a sub-buffer to become available. At the time a sub-buffer is filled with data, the Driver unblocks the Decoder, which starts reading the produced data. The Decoder is decoding sequentially all the sub-buffers that have been filled with data by Intel PT, at the same time that the hardware is writing data on subsequent sub-buffers. As soon as a Control-Flow packet is decoded, it is used in order to trace the execution of the original program. In order to

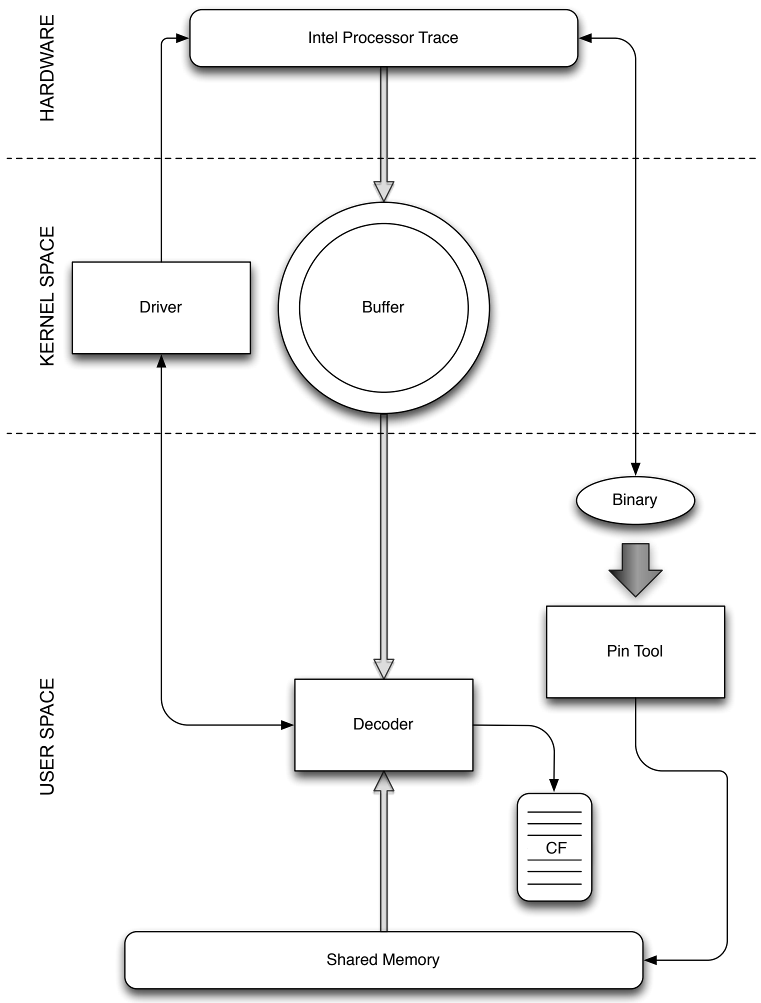


Figure 3.1: Design of the system.

do so, the Decoder requires the disassembly of the code that was executed by the Pintool inside the Code Cache. Our Pintool achieves this by disassembling all the Code Cache instructions and storing them inside a POSIX Shared Memory. By using the disassembly together with the Intel PT packets our decoder extracts the Basic Blocks executed by the original program and reconstruct the complete Control-Flow of the program.

3.1.2 Driver

The Intel PT Kernel Module is probably the most important part of our reconstruction tool, since it is responsible for controlling the Intel PT hardware feature, as well as coordinating and communicating with all the other components of the system. Our Kernel Module has been built by extending the Intel Simple-PT driver and customizing it in order to satisfy our needs. All the modifications will be described extensively on section 4.

Our Intel PT Driver is a critical part of the overall system, as it exposes Intel PT functionality to user space. This allows a user to set up and enable the tracing of a specific process, by accessing several Kernel Module parameters exposed by the Driver. A user is capable of specifying the application that desires to be traced by setting its CR3 value on the appropriate parameter. The Kernel Driver receives this update on the CR3 parameter and updates the CR3 register. Intel PT uses this CR3 register in order to decide on the process, for which it should collect Control-Flow information data. In addition, a CPL Module parameter is offered by the Driver. By setting this parameter the user is able to decide whether, Control-Flow information for the user or kernel space execution of an application will be collected. Our tool utilizes the offered functionality, in order to trace the user space execution of a Pintool. This Pintool will be the JIT environment used in order to prove that our system is able to extract the execution flow of the original program. Furthermore, the Driver can be used in order to set the IP filtering option of Intel PT. In our case, we only care about Intel PT data produced by the execution of the Code Cache, since this is where the generated JIT code resides. We have used this idea in order to reduce the amount of the produced Intel PT packets, by having PT trace only code executed inside the Code Cache address ranges. The user is also able to start or stop the Intel PT hardware feature whenever needed by setting a specific Module parameter. Whenever this parameter is set, the Driver is notified and the Intel PT hardware is controlled accordingly.

Furthermore, our Driver exposes further functionality, that has been used throughout this project. By accessing the Kernel Module a process is capable of accessing the buffer where the Intel PT data are written, wait for a sub-buffer to become available of reading and even slowdown the traced process in order to avoid overwrites. Other capabilities of the Driver will be described on the following subsections of the Design overview, depending on the component of our system that utilized them.

3.1.3 Ring Buffer

As previously mentioned, the Intel PT uses the ToPA mechanism in order to store the trace output. Our ToPA setup consists of 20 sub-buffers that are connected together, in order to create a ring buffer. This is achieved by having the last ToPA entry point to the first one, using the END bit of the ToPA entry. Our Kernel Modules configures this buffer that resides inside the Kernel space. The default Simple-PT Driver was build for offline decoding of the Intel PT packets. This means that when a process is being traced, all the PT packets will be recorded on the ring buffer. When the execution of the program is completed, the decoder will copy the whole ring buffer to the user space in order to be decoded. This implementation, however, faces a couple of limitations that are vital for our system and need to be resolved. First of all, the tracing of a program is limited by the size of the ring buffer. When the ring buffer gets full, all the new data will overwrite the old, meaning that only the latest trace output will be preserved. Secondly, copying the entire ring buffer to the user space, in order to be decoded, is costly when it needs to be done in parallel to the execution of the program. In order to address, this problems and be able to decode the PT packets in real time, we had to adjust the way the buffer is accessed by our

Kernel Driver.

The most promising solution in order to refrain from copying the data to the user space, but at the same time avoid overwrites, is to decode the PT data directly from inside the Kernel ring buffer. In order for this to be achieved, a set of steps are required. First, a way in order to know when a sub-buffer is filled with data is required. As such, the decoder can be notified and the decoding of that particular sub-buffer can take place is parallel to the programs execution. In addition, our mechanism must ensure that overwrites are avoided, when the Intel PT packets are produced faster than decoded. For this reason, we have utilized the INT bit of the ToPA Entry. As previously mentioned, if the INT bit is set for a specific ToPA output region, a notifying PMI interrupt will be raised, at the time this memory region is filled. By capturing this signal we are able to know when a sub-buffer is ready and notify the Decoder to start reading and decoding it. In addition, by monitoring the distance between the sub-buffer that PT writes and the one the Decoder is accessing, we are able to slowdown the main process in cases that an overwrite is imminent.

3.1.4 Pin Tool

As previously mentioned, Pintool is the JIT environment showcased throughout our work. In order for our system to be able to reconstruct the Control-Flow of the program, apart from the Intel PT data, some additional information about the execution of the application need to be collected. We achieve this by utilizing Pintool and sharing all the required data with the Decoder.

As aforementioned, Pin intercepts the execution of the executable and generates new code that is then placed inside the Code Cache. In order for our reconstruction technique to be performed, the Intel PT data need to be used in order to follow the code executed by Pintool. This means that, the exact code generated is required by the Decoder. Our system's approach, is to acquire the complete Code Cache disassembly through Pintool and share it with the Decoder. In order for faster decoding to be performed, we decided to create a Control-Flow table. This table contains all the transitions from one address of the program to another, and correspond to conditional branches and direct or indirect jump/ calls of the code. At the same time, information about whether each instruction is generated by Pintool or whether it belongs to the original program is stored inside this table. The Decoder can use the table in combination to the Intel PT Control-Flow decision packets in order to follow the complete execution of the Code Cache Code. This Control-Flow Table is stored in a Shared Memory region that can be accessed by the Decoder at any time.

Since we only care about the code executed inside the Code Cache, we have utilized the IP filtering functionality in order to configure Intel PT to collect tracing information only for the code that resides there. However, the Code Cache is not accessed only once and the generated code is not executed uninterrupted. Pintool's execution moves back and forth from inside the Code Cache and Pin's code. This means that, the Code Cache is being accessed at different regions each time the execution returns to it. Unfortunately, PT doesn't collect any information about the location inside the Code Cache where the control transfered, due to this access being performed by a direct jump. For this reason, information about where in the Code Cache the collected packets correspond to is required. We utilize Pin's API in order to register a callback routine that records the address where the Code Cache was accessed each time. We store the Code Cache accesses in a queue residing in the Shared Memory and being accessed by the Decoder, in order to enable the reconstruction.

Finally, after generating code for a Trace, Pin attempts to link this generated code with other segment of generated code inside the Code Cache. This is mainly performed in order to reduce the Code Cache accesses, meaning that when multiple Code Cache Traces are linked between each other, a single Code Cache will result in multiple Traces being executed. In order for the Control-Flow reconstruction to be correct, those transitions between the Code Cache Traces need to be applied on the Control-Flow Table. However, since those links where not created by Pin at the time a Trace was placed inside the Code Cache, having Pintool placing those transition inside the Control-Flow table will result in tracing the wrong code. These links need to be applied by the reconstruction tool at the exact same time that they were created when the Code Cache code was being executed. For this reason, we have created a Link table that contains information about the new transitions and when they should be applied to the Control-Flow table. This table is placed in the Shared Memory region as well and will be further described in section 4.

3.1.5 Decoder

The Decoder is the main component responsible for reconstructing the Control-Flow of the original program. It is responsible for reading the recorded Intel PT data from Kernel space, decode them and use them in order to reproduce the Control-Flow that the original program followed during its execution.

As the name of this component suggests, its main task is to decode the data collected by Intel PT. In order to achieve this the Decoder needs to have access to the Kernel Ring Buffer that was previously described. The Driver offers the option to use memory mapping in order to map the Kernel Ring Buffer in the address space of the Decoder. We use memory mapping in order to eliminate the overhead of copying the Intel PT data from Kernel space to User space. As previously illustrated, the Buffer consists of several smaller sub-buffers connected as a list. The Decoder is implemented is such a way that it decodes one sub-buffer at a time. In addition, the Decoder is allowed to read data for decoding, only if Intel PT has already finished recording on that sub-buffer. This approach was mainly followed in order to avoid writing and reading the same sub-buffer simultaneously. We accomplish this, by having the Decoder block through a Blocking I/O, until the next sub-buffer is filled with data for decoding. After a sub-buffer is filled with data, the Decoder is unblocked, enabling it to read the Intel PT data and start decoding.

The decoding is performed using the Intel Processor Trace Decoder Library (libipt) distributed by Intel. In general, for the decoding part, we use as a basis the implementation of a simple Decoder, that comes together with the Simple-PT Kernel Module. This Decoder decodes the data and reports every packet that was collected during the tracing of the program. Our system only utilizes a subset of those Intel PT packets. This means that, certain packets can be excluded at the moment they are discovered. We modify the decoding process and only focus on TNT, TIP and FUP packets, since those are sufficient for reconstructing the execution flow of the traced application.

Reconstructing the Control-Flow, requires our algorithm to follow the code that was executed in association with the Intel PT recorded packets. For this reason, and as previously discussed, our Pintool creates a Control-Flow table that includes all the Control-Flow transitions, and information about whether each one of them belongs to the original program or to the generated Pin code. The Decoder is able to access this table by attaching to the Shared Memory segment, where the Pintool has stored it. Using this transition table in combination with the decoded Intel PT packets, our algorithm is able to actually trail the Control-Flow table transitions correctly and thus extract the Basic Blocks of the original program.

One limitation we had to overcome is the fact that, during the JIT execution, changes on the generated code take place. One of the most common code changes is the linking of different Code Cache Traces to each other. This means that after these links are performed, our Control-Flow table would be rendered obsolete. Our system, in order to stay up-to-date, and follow the correct execution Flow needs, to perform those transition updates on the Control-Flow table. However, those transitions need to be updated at the correct time during the Control-Flow reconstruction. For this reason, our Pintool has created a Link table containing information about the newly created links between traces, as well as when those transitions should be added on the Control-Flow table. Since the Decoder is the component of our system that is using the Control-Flow table in order to follow the execution of the program, we decided to perform the transition updates here. More specifically, we noticed that the links occur at the moment a new Code Cache Trace is being accessed. Our Decoder is looking for those newly added Traces and by using the Link table is able to update the Control-Flow table with all the new transitions correctly.

Our Decoder is able, by combining all the information collected by our other components, to successfully extract all the Basic Blocks of the original program, leading to the reconstruction of the Control-Flow of the original program. The Decoder, as well as all the other components of the system, will be described in depth in the following section.

Chapter 4

Prototype Implementation

Our system is a prototype for hardware-accelerated Control-Flow reconstruction of applications, executing inside Just-in-Time environments. This chapter describes the implementation of the system and is structured as follows:

In chapter 4.1 the set up and enabling of Intel PT is described. Chapter 4.2 covers the details of the Ring Buffer used for the output of the trace data. Chapter 4.3 focuses on the Pintool used in order to setup a JIT environment. This Pintool is also used in order to collect some data of interest that will assist on the Control-Flow reconstruction. Finally, chapter 4.4 describes the developed Intel PT Decoder and the Control-Flow reconstruction algorithm of our system.

By combining all components, we developed a hardware-accelerated Control-Flow reconstruction system of applications executing inside a JIT. Our system leverages the novel hardware feature Intel PT and offers real time reconstruction of the original program, by extracting it from the JIT environment's execution.

4.1 Intel Processor Trace Configuration

Intel Processor Trace is the hardware feature that our system utilizes in order to reconstruct the Control-Flow of the original application, executed inside a JIT environment. Properly setting up the Intel PT hardware to trace only the code of interest can lead to huge performance benefits. By default Intel PT, traces the execution of the whole CPU, meaning that an enormous amount of data is being recorded. Decoding all those packets could result in major system overheads and slowdowns. In order for our system to work properly, only trace information for the execution of Pin's Code Cache is required. During a Pintool's execution, all the generated code is placed inside the Code Cache, which resides on a fixed memory region. Configuring Intel PT to only trace code that is located on that memory region, can greatly reduce the amount of generated data and subsequently decrease the time required for them to be decoded.

In order for the setup to be applied, we have developed an application that configures the Intel PT hardware, depending on our needs, and begin the tracing procedure. After everything is setup and Intel PT is enabled, our Pintool is spawned using Pin. Pin is executed using the fork() system call.

4.1.1 Filtering

The Intel PT filtering setup is configured as following:

CPL-Filtering

As described in section 2, Intel PT can be configured to only trace the execution of a process in the User space. Since our system focuses on reconstructing the Control-Flow of the original program, only branch decisions of the User space code is required. The Simple-PT Driver offers two Module parameters, that allow the user to specify if Intel PT will trace the Kernel or User space execution. By setting the Kernel parameter to 0 the Driver is instructed to set CPL to greater than 0, thus enabling User space tracing.

CR3-Filtering

CR3 filtering is used by Intel PT to trace only a specific process, instead of the entire system. In order for Intel PT to trace only a single CR3 value, software has to write the CR3 to the IA32_RTIT_CR3_MATCH MSR. For this reason, we have created a Module IOCTL that can be used to pass a process's PID to the Driver. When this IOCTL is called, our Driver discovers the CR3 value of the process using this PID and updates IA32_RTIT_CR3_MATCH MSR. The Driver is able by retrieving the task_struct of a process, using its PID, to collect its CR3 value. The IA32_RTIT_CR3_MATCH MSR is then set with the acquired CR3 value and Intel-PT is able to trace the Pintool correctly.

As described above, we have developed an application responsible for setting up Filtering and enabling PT. Our initial idea was to configure the CR3 value before the Pin was executed. However, we noticed that Pin's PID value differs from the Pintool's one. This means that if we configure Intel PT to trace the process using the PID of Pin, our tool won't be traced, since the Driver will set up the wrong CR3 value. Since it is essential to follow the execution of Pintool's Code Cache, we need to acquire the correct PID, in order for Intel PT to discover the correct CR3 value. To achieve this, we call the IOCTL responsible for setting up the CR3 value, from within the main function of our tool.

IP-Filtering

As seen in section 2, IP filtering sets Intel PT to only trace certain regions of the codes execution. To enable IP-Filter ranges, software has to modify two MSRs. These MSRs are IA32_RTIT_ADDRn_A and IA32_RTIT_ADDRn_B MSR, where *n* represents the identifier of the given IP filter. In general, Intel PT allows two IP-Filter ranges to be set. We have utilized this functionality, in order to focus the tracing only to the Code Cache execution, instead of the whole system. Our system is interested in reconstructing the Control-Flow of a program executed in a JIT environment. This means that the Code Cache needs to be traced, since this is where the generated JIT code is placed. The Simple-PT Driver offers two Module parameters, allowing the user to specify the ranges that Intel PT will collect Control-Flow packets. Using these parameters, the Driver is able to configure the MSRs previously described. We set up the Code Cache address ranges from within our application responsible for all Intel PT initializations.

4.1.2 Return Compression

When Return Compression is enabled, TIP packets for near return (RET) instructions will be compressed by Intel PT. If the RET target matches the next IP of the corresponding CALL, then the TIP packet is obsolete, since the decoder can deduce the target IP by maintaining a CALL/RET stack of its own. In order to minimize complexity we disable Return Compression, by accessing the corresponding Module parameter offered by the Driver. The Driver then sets the proper MSR bit in order for the Return Compression to be disabled.

4.1.3 Processor Affinity

Processor affinity, or CPU pinning, enables the binding and unbinding of a process or a POSIX thread to a certain CPU. That process or thread will execute only on the designated CPU or CPUs rather than any CPU. Our process responsible for setting up Intel PT and spawning the Pintool, utilizes this option by calling `sched_setaffinity`, in order to attach Pintool on a specific core using its PID. Since our Pintool is executed via fork() system call, it inherits its parent's CPU affinity mask. This way the Decoder can focus on decoding trace data, only produced by a certain CPU core. Pining the Pintool or a specific core makes the decoding process less complicated, since all the data will be written on the same Kernel Ring Buffer. In addition, the Pintool will execute on only one of the CPU cores, allowing the remaining to be utilized by other processes.

4.1.4 Trace Enabling

Finally, after all the initializations are performed, the Intel PT hardware is enabled for tracing. This is done through another Module parameter. When the value of this parameter is set to 1, the Driver is notified and the TraceEn bit of IA32_RTIT_CTL MSR is set to enable PT tracing.

4.2 Ring Buffer

The Kernel Ring Buffer, where Intel PT records the trace data, is an essential part of our whole system. We utilize the Table of Physical Addresses, described on Chapter 2.2, in order to facilitate the implementation of long-term Intel PT tracing. The ToPA mechanism provides a hardware-assisted mechanism to notify software if a configured ToPA entry has been filled, by using the ToPA Entry Field INT. In addition, it offers the option to create a ring buffer by having the last entry point to the first one, through the ToPA Entry Field END. By utilizing all these capabilities, it is possible to implement a reliable long-term tracing mechanism. We leverage the above techniques, in order to create the following ToPA configuration during usage.

4.2.1 ToPA Configuration

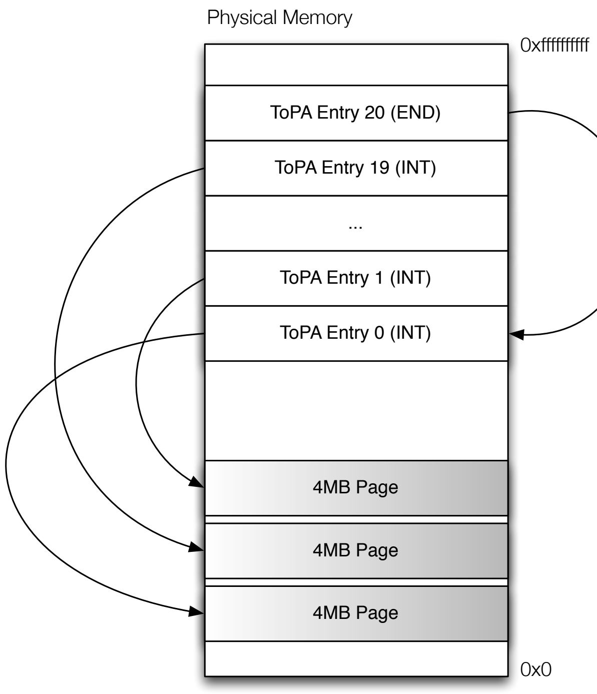


Figure 4.1: ToPA Configuration.

Our system requires to be able to decode data in parallel to recording. For this reason we have utilized the ToPA mechanism to create a Ring Buffer, consisting of several sub-buffers. This way, the Decoder can read the data residing on already filled sub-buffers, while the Intel PT hardware is writing its trace data on the remaining free ones. Our configuration consists of 20 sub-buffers connected to each other through the ToPA mechanism. Each sub-buffer has been allocated a Physical Page of size 4MB. As shown in Figure 4.1, the first 20 ToPA Entries are pointing to 20 different Physical Pages and have the INT ToPA Field set. ToPA offers the option to create a circular buffer by having the last ToPA Entry point to the first one. The last ToPA Entry is configured by setting the END ToPA Field. As we can see, our last ToPA Entry is pointing to the ToPA Entry 0, thus creating a Ring Buffer.

4.2.2 PMI Signal Handler

As previously, mentioned we utilize the INT Field of the ToPA Table Entry, in order to be informed whenever a sub-buffer is filled with data. By setting the INT Field on each one of the ToPA Entries, an LVT PMI is automatically activated during the initialization and configured as a Non-Maskable Interrupt (NMI). The NMI is a hardware interrupt, of the highest-priority, capable of interrupting all software and non-vital hardware devices. This interrupt cannot be ignored through the use of interrupt masking techniques.

In order to be able to know when a sub-buffer is full of trace data, we have implemented a PMI NMI Signal handler. We distinguish this PMI from others by checking bit 55 (Trace_ToPA_PMI) of the IA32_PERF_GLOBAL_STATUS MSR. In addition, in order to make sure that the hardware is writing on the next buffer, than the one we received the signal for, we check bits 31:7 of the IA32_RTIT_OUTPUT_MASK_PTRS, which indicate the entry of the table that is currently in use. This way, each time a PMI Signal is received, we can be sure that a specific sub-buffer is full and we are able to perform certain actions.

4.2.3 Ring Buffer Overwrites

One problem that may arise while using a Ring Buffer, may be the case where the writer is quite fast and may catch up with the reader. If this happens, trace data that haven't been decoded yet will be overwritten with new, leading to Intel PT packet loss. Consequently, we need to implement a way that handles this case. We have utilized the PMI Signal handler in order to create a mechanism for avoiding overwrites. More specifically, each time a PMI signal is received, we check the distance between the writer and the reader. We know the position of the writer by using the IA32_RTIT_OUTPUT_MASK_PTRS, mentioned before. In addition, the Kernel Driver holds the offset of the sub-buffer that the Decoder is currently reading. By monitoring the distance, between the writer and the reader inside the PMI Signal handler, we are able to pause the Pintool that Intel PT is tracing, when an overwrite is approaching. This way, no trace data are recorder on the Ring Buffer, thus avoiding any overwrite of Intel PT data. We pause the Pintool by sending a SIGSTOP signal using killpid(). Each time a PMI signal is received, the Signal handler is checking whether the reader has obtain a sufficient distance from the writer. If so, the Pintool is unpause by sending a SIGCONT signal.

4.3 Pin Tool

This work focuses on extracting the execution of an application from inside a JIT environment and at the same time reconstructing its Control-Flow. As previously described, the JIT environment we have used throughout our work is Pin. When an application is executed by a Pintool, Pin generates a new sequence of code depending on the Basic Blocks of the original application being instrumented. Our system is attempting to isolate the original code from the one generated by Pin, in order to extract the original Basic Blocks. In order to do so, our system is using branch decision information collected by the hardware to follow the execution of the generated code placed inside a specified memory region, the Code Cache. However, certain information are not collected by the hardware, meaning that they need to be acquired in a different way. We collect the required information from inside the Pintool, by utilizing the Pin API. This way our system is able, by combining the Control-Flow decisions collected by Intel PT and the complementary information by Pintool itself, to reconstruct the actual Control-Flow of the original application that is executing inside the JIT environment.

4.3.1 JIT Generated Code

Table 4.1 displays how a Trace of the original program is transformed to a Code Cache Trace.

Original Trace	
Address	Instruction
82aacb5	test ...
82aacbb	jz 82aacc4
82aacbd	or ...
82aacc4	mov ...
82aacca	test ...
82aacbd	or
82aaccd	jz 82aacd9
82aaccf	or ...
82aacd9	and ...
82aacde	cmp ...
82aace3	jz 82aacfd

Code Cache Trace		
CC Address	ORIG Address	Instruction
f5a371e8	82aacb5	test ...
f5a371ee	ffffffff	data16 nop
f5a371f0	ffffffff	jz f5a76dc0
f5a371f6	82aacbd	or ...
f5a371fd	82aacc4	mov ...
f5a37203	82aacca	test ...
f5a37206	ffffffff	data16 nop
f5a37208	ffffffff	jz f5a76ddc
f5a3720e	82aaccf	or ...
f5a37218	82aacd9	and ...
f5a3721d	82aacde	cmp ...
f5a37222	82aace3	jz f5a76df8
f5a37228	ffffffff	jmp f5a76e14

Table 4.1: Pin generated code. On the left hand the original Trace is shown, while on the right hand the generated code for this original Trace appears.

4.3.2 Code Cache Trace Discovery

An important step in reconstructing the Control-Flow of the original program is to follow the execution of the Code Cache. This means that, we need to be able to know exactly where on the Code Cache code, the collected packets correspond to. As previously described, the Intel PT hardware has been set to only trace the execution of the Code Cache, using the IP-Filtering mechanism. In general, Pin uses a direct jump in order to enter the Code Cache. This means that, no TIP packet containing the address of the Code Cache Trace, that will be executed, is recorded by Intel PT. By observing Pintool's execution flow we found out that, before entering the Code Cache, the target of the direct jump is modified in order to point to the right Code Cache Trace. This direct jump lies at the same address and is reached by an indirect jump. Subsequently, a TIP with the exact same address is received each time the execution of the Pintool moves to the Code Cache. This TIP is quite helpful and will be referred to as Code Cache Entry TIP. This way, the decoder is able to know that the execution of a new Code Cache Trace is taking placing and perform the reconstruction of this original Trace.

However, by using this technique we are unable to know which Code Cache Trace was executed each time. In order to overcome this limitation, we have created a queue containing the Code Cache Trace addresses in the exact order the control of the Pintool has accessed them. This queue is stored in a Shared Memory segment. Our reconstruction algorithm is therefore able, every time a Code Cache Entry TIP appears, to know which Code Cache Trace the subsequent collected Intel PT packets refer to.

Pintool utilizes the callback routine CODECACHE_AddCodeCacheEnteredFunction, which adds a function that gets called whenever control enters the Code Cache. Using this function, we have the opportunity to insert the Code Cache Trace address in the queue. The Decoder is able to know which Code Cache Trace is traced, every time a Code Cache Entry TIP appears, by getting the first element of the queue. Using this queue together with collected Intel PT packets, as well as all the structures described in this section, the Decoder is able to reconstruct the Control-Flow of the original application.

4.3.3 Control-Flow Table

The main idea behind our system is to reconstruct the Control-Flow of the original application executed inside the JIT environment. In order to do so, our reconstruction algorithm, that will be described later, requires to follow the exact execution of the Code Cache code together with the collected Intel PT packets. Our initial idea was to store the complete Code Cache code in a Shared Memory, so that it could be accessed and used by our reconstruction tool. Our tool would then be able, by accessing the disassembly of a particular Code Cache Trace, to follow the code instruction by instruction. Each time a conditional branch or an indirect jump/ call would be found, the next Intel PT packet would be used, in order to discover the next address that Pintool accessed. In addition, each disassembled instruction would be marked as original or generated by Pin. This way our algorithm would be able to differentiate the original Basic Blocks from those of the generated JIT code, and isolate them.

However, this approach suffers from certain drawbacks. First of all, it requires the whole disassembly of the Code Cache code to be written in memory. Depending on the size of the main application, this may demand a lot of memory to be used. In addition, this approach may be quite slow, since the reconstruction algorithm would need to parse every instruction of the Code Cache, although it only requires a subset of those. More specifically, instead of parsing every instruction, the reconstruction algorithm only needs to know the entry of a Basic Block and the transition to the next one. For this reason, we decided that the best approach is to create a Control-Flow Table, containing the transitions from one Basic Block to the next one. In addition, we have used this table in order to store information about the original Basic Blocks that reside inside the Code Cache. This way the Decoder will be able to correlate each transition to the original Basic Blocks, and be able to extract them faster. For this reason, our Control-Flow Table contains all the discovered transitions of the Code Cache code, as well as information about whether an instruction belongs to the original program or is Pin generated.

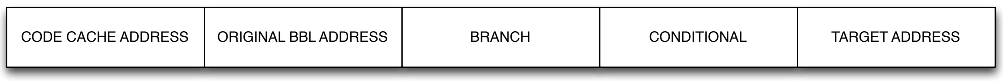


Figure 4.2: Control-Flow Table Entry Overview.

The format of the Control-Flow Table entries is shown in Figure 4.2. Each Code Cache Entry encodes a transition from one Code Cache Basic Block to another or the address of an original Basic Block. Field 1,2 and 3 are unsigned integers containing hexadecimal values that correspond to addresses. The remaining fields are integers indicating the type of the entry. The individual fields are the following:

CODE CACHE ADDRESS
Each entry of the Control-Flow table is created depending on an instruction of the Code Cache. This field indicates the address inside the Code Cache, where the corresponding instruction resides.

ORIGINAL BBL ADDRESS
This field contains information about whether an original Basic Block was discovered. More specifically, whenever the first instruction of an original Basic Block is found inside the Code Cache, its original address is written on this field. In case this instruction is also a branch instruction, the remaining fields are used in order to store the transition information. If the instruction is not a branch instruction, only the CODE CACHE ADDRESS field is set and the remaining fields are not set.

BRANCH
The BRANCH field indicates whether this entry is a transition or not. If BRANCH is set to 1, this entry field corresponds to a branch instruction that was found inside the Code Cache. If this field is set to 0, the entry is not a transition and can only contain the address of an original Basic Block.

CONDITIONAL
The CONDITIONAL field can only be set to 1 if the BRANCH is set as well. It is used in order to identify a transition as conditional or not. If CONDITIONAL is set to 1, this specific entry corresponds to a conditional branch instruction and should be handled accordingly. If set to 0 while BRANCH is set to 1, the entry corresponds to a direct or indirect branch instruction of the Code Cache. The Decoder can use this field in order to be able to know how this transition needs to be handled, at the time the reconstruction takes place.

TARGET ADDRESS
In case that an entry refers to a Control-Flow transition, this field contains the target address of the branch instruction. This field is not set, if the corresponding instruction of the Code Cache is an indirect branch, as the address is unknown at the time the Control-Flow Table was created.

4.3.3.1 Code Cache Disassembling

The first step of filling the Control-Flow Table, requires discovering all the branch instructions that reside inside the Code Cache. In order to do so, we need to be able to check each instruction of the generated code, and use the ones we need to create the table. For this purpose, we have used the callback routine CODECACHE_AddTraceInsertedFunction, which adds a function that gets called whenever a new Trace is placed inside the Code Cache. In addition, this callback routine supplies us with the complete Trace, for which the callback was invoked. Subsequently, this is the proper place to acquire the disassembly of a Code Cache Trace. We achieve this by using XED, X86 Encoder Decoder, provided by Intel. XED comes with an API enabling us to perform disassembly on any given x86 instruction. By iterating through every instruction of the Trace, we are able to obtain the complete disassembly of the generated code that is getting placed inside the Code Cache.

During the Control-Flow Table's creation, we identified some some cases where the Code Cache was not accessed at the beginning of a Code Cache Trace. Instead, the control of the program moved at some Code Cache regions where Pin generated code resided. These regions, mainly consist of small code sequences that end with a jump to a proper Code Cache Trace. Due to the reason that, these code sequences do not belong to a Trace, that was placed inside the Code Cache at some point, the above described disassembly procedure was never performed for them. This means that, we do not own their disassembled code, rendering us unable to follow the execution of the Code Cache every time it moves there. The approach we have followed in order to overcome this problem, is to acquire the disassembly for these code sequences at a different point than this of the Code Cache Traces. We have used CODECACHE_AddCodeCacheEnteredFunction callback routine, which gets called whenever the control is entering the Code Cache. However, this callback routine does not return the complete Code Cache Trace, contained at the address where the Code Cache was accessed. Instead, it only supplies us with the address where the Code Cache was entered. In order to obtain the disassembly for this code sequence, we begin by disassembling the instruction at the given Code Cache entered address. XED API, provides us with a function that returns the length of an already disassembled instruction. Having the length of the disassembled instruction at the address where the Code Cache was entered, we are able to iterate the address offset to the next address of the code sequence. Performing this procedure for every instruction until we find a padding instruction, we are able to obtain the whole disassembled code for these Code Cache code sequences. In our case, the padding instruction that ends the disassembling procedure is an add byte ptr [rax], al instruction. Due to the reason that, these Code Cache code sequences do not contain a Trace, we do not need to know whether each instruction is an instruction of the original program or is Pin generated.

4.3.3.2 Control-Flow Transitions

After the disassembly of the code executed inside the Code Cache is collected, the Control-Flow Table creation can be performed. As described, this table acts as a transition table, allowing the Control-Flow reconstruction algorithm to be able to follow the execution without parsing the disassembled code instruction by instruction. Storing all the transitions of the executed code inside the table, involves discovering the branch instructions and identifying their type. As previously described, we have utilized XED in order to disassemble the Code Cache Code instruction by instruction. At the moment a new instruction is disassembled, our Pintool checks whether it is the case of a branch instruction. If a branch instruction is found, it is added on the next free cell of the Control-Flow Table. Depending, on the type of the branch instruction discovered, the appropriate fields are set on the new Control-Flow Table entry. The types of branch instructions and their setup on the table is the following:

Direct Calls/ Jumps
Direct calls and jumps are inserted on the table as a entry with the BRANCH field set to 1 and CONDITIONAL set to 0. The TARGET field points to the destination of the call/ jump instruction.

Indirect Calls/ Jumps
Indirect calls and jumps are inserted on the table as a entry with the BRANCH field set to 1 and CONDITIONAL set to 0. The only difference with an entry created due to a direct call/ jump instruction is that, the TARGET field is not set.

Conditional Branches
Conditional branches are inserted on the table as a entry with the BRANCH field set to 1 and CONDITIONAL set to 1. The TARGET field points to the destination of the branch instruction.

The address of each branch instruction, that is found inside the Code Cache, is collected using TRACE_CodeCacheAddress of the Pin Code Cache API, and is stored on the CODE CACHE ADDRESS field of each entry.

4.3.3.3 Original Basic Blocks

An important step of reconstructing the Control-Flow of the application, is this of the Decoder being able to know when an original Basic Block is found in the generated code. As described, we have included a field in the Control-Flow Table, that allows us to store the original Basic Blocks. Therefore, when parsing the disassembly instruction by instruction, instead of only caring about the branch instructions, we also focus on discovering the Basic Blocks of the original application.

For this reason, we have used TRACE_Address, offered by Pin's Trace API. The TRACE_Address function returns the original address of an instruction residing inside the Code Cache. In case this instruction is Pin generated, and doesn't belong to the original application, this function returns 0xFFFFFFFF. Using this information, our Pintool is able to differentiate an instruction between original or Pin generated. In order to find out the Basic Block addresses of the original program, we need to be able to get the first original address of the Basic Block. In general, Pintools generated code follows a certain format. An original Basic Block is transformed in such a way that, its exit instruction is replaced by a branch instruction generated by Pin. This means that, each Basic Block of the Code Cache Trace may contain only a single original Basic Block. We have used this knowledge to extract the original Basic Block included inside a Code Cache Trace Basic Block. Starting from the first instruction of a Code Cache Basic Block we iterate until we find the first instruction that TRACE_Address returns an original address for. We consider this to be the first instruction of an original Basic Block and thus the original Basic Block address.

In order to make sure that the acquired address is correct, we need to be able to compare it to the actual original Basic Block. For this reason, we have configured our Pintool in such a way that it collects all the Basic Blocks for each original Trace beforehand. This is performed using the callback routine TRACE_AddInstrumentFunction, which adds a function that returns each TRACE of the original application, before being used by Pin in order to generate the Code Cache Trace. Our Pintool stores every Basic Block for each Trace inside an internal structure. This way every time a new original Basic Block is discovered, using the previously described technique, our Pintool is able to know if the acquired address is correct or not, by checking the Basic Blocks included in the corresponding original Trace. If the discovered original Basic Block is contained inside the original Trace, we insert it on the Control-Flow Table on the next free cell. The BRANCH, CONDITIONAL and TARGET fields of this Control-Flow Table entry are set to 0.

4.3.3.4 Special Cases

Identifying the original Basic Blocks inside a Code Cache Trace, as described, is simply performed by following the Code Cache code, instruction by instruction, and storing the original Basic Blocks and the branch instructions. However, there were certain cases where the original Basic Blocks were modified in such a way, that following the described method of discovering them could not work. These cases, and the approach we followed in order to correctly insert them on the table, are the following:

Wrong Basic Blocks

Whenever a REP instruction appears on the original code, Pin is generating code in order to handle it. However, the generated code makes our tool report some wrong Basic Blocks, since the newly created Code Cache Blocks don't comply with how our algorithm is inserting the original Basic Blocks to the Control-Flow Table. When a REP instruction exists inside an original Basic Block, this Basic Block is split in two Code Cache Basic Blocks by Pin. The first Block contains all the instructions until the REP and is followed by Pin generated code. The last instruction of this Code Cache Basic Block is a Pin generated branch instruction. The second Basic Block starts with the instruction the REP was being applied to and then more JIT generated code follows. Again this Basic Block is ended with a generated branch instruction. In addition, a new Trace is inserted in the Code Cache. This Trace contains only one original instruction, with that being the instruction to which the previous REP was applied. We handle this case by not creating an entry for the wrong Basic Block that appears when the original Basic Block is split into two. We only create an entry for the instruction that was issued with REP, when this appears on a new Trace by itself. We observed, that Pintool's execution inside the Code Cache moves to this newly created Trace when the REP instruction needs to be executed. This way we capture the correct original Basic Block.

Missing Original Address
When Pin generates a Code Cache Basic Block based on an original Basic Block, it replaces the last instruction with its own branch instruction. However, there are some Basic Blocks that only contain a single instruction. This happens when the first instruction of the original Basic Block is a branch instruction or a REP. When these single instruction Basic Blocks are used to generate a Code Cache Basic Block, this instruction is replaced with a Pin generated branch. Thus, when creating the Control-Flow Table our Pintool is not able to acquire the original address for this Basic Block. We fix the missing original address, by patching the missing address on the Control-Flow Table. This is performed by using the original Trace. Every time a single instruction Basic Block appears on the original Trace, we keep its address and patch it when the new Pin generated branch instruction is inserted on the table. This way the missing original Basic Block is placed correctly inside the Control-Flow Table and can be used by the Decoder in order to discover it.

4.3.3.5 Traversing the Control-Flow Table

Since creating and populating the Control-Flow Table has been performed, a way to access the information that it contains needs to be implemented. The Decoder requires the information of the Control-Flow Table in order to follow the execution of Pintool inside the Code Cache and at the same time extract the original Basic Blocks. The Decoder, starting from a Code Cache address, needs to traverse certain table entries, based on the Intel PT data. Therefore, the main functionality required, is a way to access the entry of a specific Code Cache address.

For this reason, we have created a Hash table containing the position of each Code Cache Trace inside the Control-Flow Table. Each time a new Code Cache Trace is parsed by the Pintool, and entries for it are created on the Control-Flow Table, the position of its first entry is placed inside the Hash table. This value is placed inside a cell of the Hash table, using as a key on the Hash function the hexadecimal address of the specific Code Cache Trace. The used Hash function is a modulo operation with a big prime number. Therefore, each time the Decoder needs to access the entry of a specific Code Cache Trace, it just needs to use the Address of this Code Cache Trace as a key and access the Hash Table. The returned value is the position of the first entry of that Code Cache Entry, inside the Control-Flow Table.

In the case where the Decoder wants to access an address that it's not the beginning of Code Cache Trace, the Code Cache where this address is inside needs to be discovered. In order to achieve this, our Control-Flow Table search function uses the address as a key to the Hash function and accesses the corresponding cell of the Hash table. If this Hash table entry is empty, it means that either this address doesn't exist or is inside a Code Cache Trace, that we need to find. We achieve this by performing a search on the Hash table. Starting from the returned Hash table cell, we traverse the table backwards until we find an entry. The first occupied cell may be the position of the Code Cache Trace in the Control-Flow Table, containing our address. We then perform a search for this specific address inside this Code Cache Trace. Starting from the first entry of this Code Cache Trace, we traverse every entry of the Control-Flow Table until the address we want, or an address bigger that the one we are looking for, is found. If the entries for this Code Cache Trace are over and the address or a greater one is not found, we can assume that the requested address doesn't exist inside the table. If the address we are looking for is discovered as an entry, it means that it is the start of a Basic Block, a branch instruction or both. In case a greater hexadecimal address is found, it means that the address we are looking for was not the start of a Basic Block or a branch, so it is not important to our reconstruction algorithm. Using the entry of the first greater address, gives us the first important instruction after the one we are looking for. Following the execution of the Pintool starting from this point gives the correct followed flow.

Using the above described functionality, our Decoder is able to access the entries of a specific Code Cache Trace inside the Control-Flow Table. In addition, if a Code Cache Trace is not accessed at the beginning but on a different address, the Decoder is able to start examining the entries of the Code Cache Trace after the correct Code Cache address. Combining this functionality to the Intel PT packets, allows our reconstruction algorithm to easily follow the execution of Pintool inside the Code Cache and extract the Basic Blocks of the original program.

4.3.4 Link Table

As previously mentioned, in order to reduce the number of accesses to the Code Cache, Pin is performing linking operations between different Code Cache Traces. In particular, when a new Trace is getting placed inside the Code Cache, links are created to and from other Code Cache Traces. These links are mainly created based on the relationships of the original Basic Blocks these Traces consist of. For example, two Traces linked one after another may contain Basic Blocks that are consecutive in the execution of the program. The links are created by adding a conditional or direct jump at the end of a Trace, pointing to the start address of the next Trace to be executed. Therefore, following the execution of the Code Cache, in order to reconstruct the control flow of the original program, also involves the efficient tracing of the linked Code Caches. Code Cache API provides the callback routine `CODECACHE_AddTraceLinkedFunction`, which adds a function that gets called whenever a trace is getting linked. By using this callback, we are able to discover the Code Cache address at which a link was placed, as well as the address of the Trace it was linked to.

However, directly updating the Control-Flow Table with the links at the time they appear, would possibly cause our reconstruction tool to follow a wrong path of Traces. This is mainly happening due to the reason that, links between Traces are created after a new Trace is placed inside the Code Cache. Until that point, all the other Traces that are going to be connected to this Trace, do not contain a jump to the Code Cache address of the new Trace. Our main problem lies to the fact that, the reconstruction of each Trace takes places at the time the appropriate Intel PT packets are collected. During this time, Pin may have placed new Traces inside the Code Cache, resulting in newly created links between them. Following the execution of a Trace using the collected Intel PT packets together with the links that may have been backpatched to the Control-Flow Table, would result in wrongly visiting Traces that Pin did not traverse. Thus, the collected Intel PT packets would not correlate to the disassembly of the Code Cache, leading in a wrong control flow reconstruction. This indicates that, the update of a transition link to the Control-Flow Table, should take place at some later point. We decided that the links to and from a specific Code Cache Trace should be inserted on the Control-Flow Table, at the time this particular Code Cache is being traversed for the first time by the reconstruction tool. This would ensure that, up to that point a Code Cache Trace would not be visited due to links, as it supposedly would not exist. On the other hand, from that point onwards, any link to this Trace would result in correctly following its execution. In order to achieve this, we have correlated every discovered link by Pintool, to the next Code Cache Trace that the execution of Pin is visiting. We have stored these links to a Shared Memory Hash table, using the corresponding Code Cache Trace address as a key. This way, we are able to perform the update of the links, at the time that the reconstruction tool begins the reconstruction of a particular Code Cache Trace for the first time. Therefore we ensure that, links to already visited Traces will exist, while links to not yet discovered Traces will not appear, leading to correctly replicating Pin's execution flow inside the Code Cache.

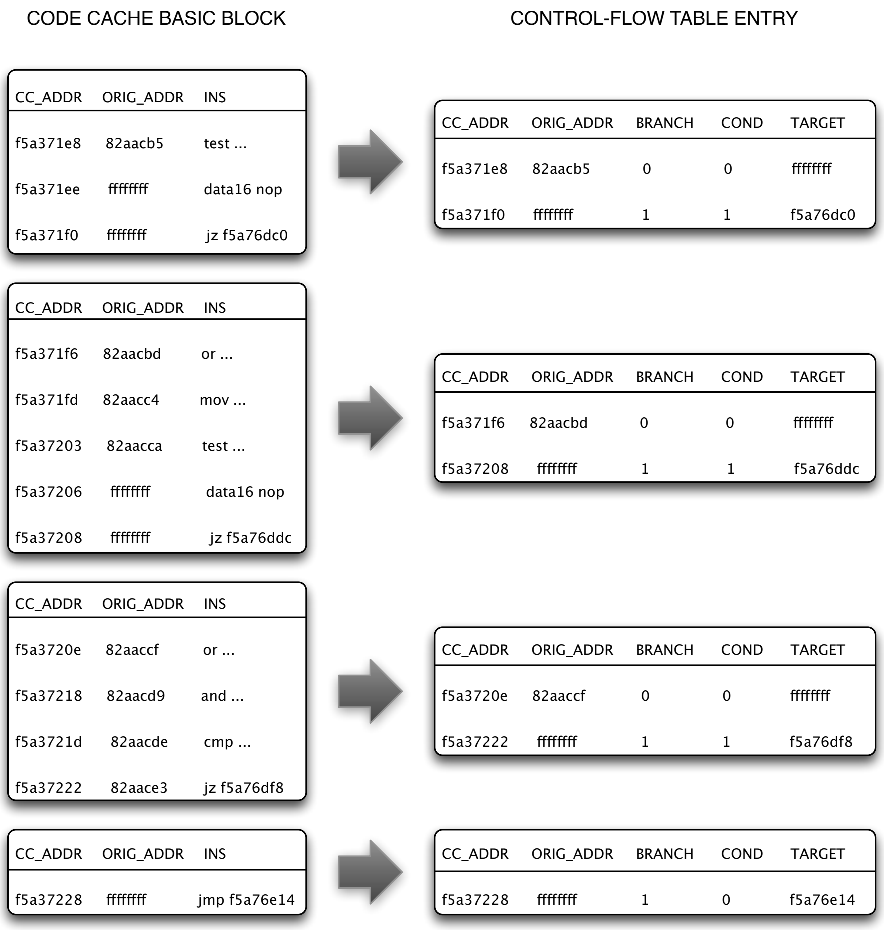
4.3.5 Usability

Figure 4.3 presents the entries that were inserted in the Control-Flow Table for a specific Code Cache Trace. We can observe that the last instruction of each Basic Block has been stored inside the table. In addition, the address of the original Basic Block included on a Code Cache Basic Block has also been written on the table. In the case where a Code Cache Basic doesn't contain original code, only its exit instruction has been stored. This way, we can easily know the original Basic Block of a Code Cache Basic Block, as well as the transition to the next Block that Pintool executed.

The Control-Flow Table is being accessed and updated by our Pintool throughout its execution. Information about every new Trace, that is being placed inside the Code Cache are stored in this Control-Flow Table. In addition, all the links that are performed by Pin between Traces, are inserted on the table by updating the proper entries. Our Decoder utilizes this table and its functionality, together with all the collected Intel PT packets, in order to follow the execution of Pintool inside the Code Cache and collect the Basic Blocks of the original application.

4.4 Decoder

The Decoder is the main component responsible for reconstructing the Control-Flow of the original program. It is responsible for reading the recorded Intel PT data from Kernel space and decoding them. Using these decoded Intel PT packets together with the Control-



CC_ADDR	ORIG_ADDR	INS
f5a371e8	82aacb5	test ...
f5a371ee	ffffffff	data16 nop
f5a371f0	ffffffff	jz f5a76dc0

CC_ADDR	ORIG_ADDR	BRANCH	COND	TARGET
f5a371e8	82aacb5	0	0	ffffffff
f5a371f0	ffffffff	1	1	f5a76dc0

CC_ADDR	ORIG_ADDR	INS
f5a371f6	82aacbd	or ...
f5a371fd	82aacc4	mov ...
f5a37203	82aacca	test ...
f5a37206	ffffffff	data16 nop
f5a37208	ffffffff	jz f5a76ddc

CC_ADDR	ORIG_ADDR	BRANCH	COND	TARGET
f5a371f6	82aacbd	0	0	ffffffff
f5a37208	ffffffff	1	1	f5a76ddc

CC_ADDR	ORIG_ADDR	INS
f5a3720e	82aaccf	or ...
f5a37218	82aacd9	and ...
f5a3721d	82aacde	cmp ...
f5a37222	82aace3	jz f5a76df8

CC_ADDR	ORIG_ADDR	BRANCH	COND	TARGET
f5a3720e	82aaccf	0	0	ffffffff
f5a37222	ffffffff	1	1	f5a76df8

CC_ADDR	ORIG_ADDR	INS
f5a37228	ffffffff	jmp f5a76e14

CC_ADDR	ORIG_ADDR	BRANCH	COND	TARGET
f5a37228	ffffffff	1	0	f5a76e14

Figure 4.3: A Code Cache Trace inserted on the Control-Flow Table. Each Basic Block of the Code Cache Trace is shown on the left side, while on the right the entries created in the Control-Flow Table appear.

Flow Table, previously described, the Decoder succeeds in following the execution of the original application inside the JIT environment and reconstructing its Control-Flow. The following subsections describe its implementation and our complete reconstruction algorithm.

4.4.1 Intel PT Data Reading

In order for the whole reconstruction to be performed, the Decoder needs to get access to the trace data, recorded by the Intel PT hardware component. As previously described, Intel PT utilizes the ToPA output mechanism in order to write the trace data directly to physical memory. The whole ToPA mechanism and the memory that is utilized by it, are being configured by our Kernel Driver during its initialization. Therefore, all the Intel

PT data are written in a kernel Buffer that needs to be accessed by our software Decoder. Mapping addresses which are in the Kernel virtual mapping into User space is straight forward. We have implemented a map method for the Driver using remap_pfn_range(), to map the physical address of the Kernel Buffer to a User space address. This function can be called from our Decoder using the mmap() system call to the Driver. This way the Kernel Buffer, where the Intel PT are written, is directly mapped to the address space of our User space Decoder. This enables our tool to directly read the data from inside the Kernel Ring Buffer, without any copies from Kernel to User space to be required.

As previously described, our Kernel Ring Buffer consists of 20 different sub-buffer. The main idea is to allow the Intel PT hardware to write the trace data on the buffer, while the Decoder is reading data from the already filled sub-buffers. For this reason, the Decoder needs to wait for a sub-buffer to become available before it starts reading. This is implemented by developing a blocking read function on the Kernel Driver. Each time our implementation of read() is called, the process calling it is blocked by using a Blocking I/O operation, through a call to wait_event_interruptible(). At the moment the next sub-buffer becomes available, a signal is received by our PMI signal handler and the Decoder is unblocked for reading. This way the Decoder is reading only the sub-buffers that the Intel PT hardware has filled with trace data. In addition, a sub-buffer is never accessed by the reader and the writer at the same time, thus avoiding coherency and synchronization problems.

4.4.2 Intel PT Data Decoding

Our Intel PT data Decoder is based on the test Decoder that comes together with Simple-PT. Simple-PT's Decoder utilizes libipt, which is a general-purpose Intel PT decoding engine. Our system only requires information about the executed control flow, whereas Intel PT also provides execution information such as TSC, MNT and CBR. For our use, it is sufficient to focus on flow-information packets only. Since we only consider TNT and TIP packets, it seems as a good idea to exclude all the remaining Intel PT packets from the decoding process. However, the decoding process itself is required to determine the offset to the following packet, since Intel PT packets do not have a uniform packet size. Therefore, we can't avoid decoding all the packets but we can only focus on TNT and TIP packets, while all the other packets are not further analyzed. The buffer containing the Intel PT trace data is decoded into a sequence of Intel PT packets. Only a small subset of Intel PT packets are considered and handled.

4.4.3 IP Compression

The IP payload in a TIP.FUP, TIP.PGE, or TIP.PGD packet can vary in size, based on the use of IP compression. IP compression is an optional compression technique the processor may choose to employ to reduce bandwidth. With IP compression, the IP to be represented in the payload is compared with the last IP sent out via a TIP packet. If that previous IP had the same upper (most significant) address bytes, those matching bytes may be suppressed in the current packet. The processor maintains an internal state of the

“Last IP” that was encoded in trace packets. Our Decoder employs the same technique in order to fill out the compressed bytes. We keep a "Last IP" field that holds the address of the last TIP that was received. This stored address is not compressed, as each time a new TIP appears, we complete the missing bytes using the previous value of the "Last IP". This leaves the Decoder with only uncompressed addresses.

4.4.4 Code Cache Entry TIP

As previously described, each time the execution of Pintool enters the Code Cache we received a TIP containing a specific address. This Code Cache Entry TIP allows the Decoder to know that all the subsequent Intel PT packets refer to the execution of Pintool inside a Code Cache Trace. We discover the address of this Code Cache Entry TIP, by ignoring all the Intel PT packets until we receive a TIP refering to the execution of the Code Cache. The first TIP that is inside the Code Cache address ranges, is the Code Cache Entry TIP. This address is stored and used by the Decoder in order to discover the TIP packets that refer to a Code Cache Trace being accessed by the execution of Pintool.

However, since no TIP packet for a specific Code Cache Trace address is collected, the Decoder is not able to know the exact Code Cache Trace the Intel PT packets refer to. In Section 4.3, we present the complete procedure that Pintool follows in order to create a queue, of the exact order, that the Code Cache Traces were accessed. Our Decoder utilizes this Trace queue, in order to be able to discover the Code Cache Trace, that the Intel PT packets refer to. More specifically, each time a Code Cache Entry TIP appears, the Decoder gets the first element of the queue. Since all the Code Cache Trace addresses are inserted on this queue by Pintool, and a TIP packet is received each time a Code Cache Trace is accessed, it is safe to assume that the first element of the queue will be the address of the Code Cache Trace that a Code Cache Entry TIP refers to. Our Decoder uses the discovered Code Cache Trace address, in order to access the Control-Flow Table at the correct position and start the reconstruction of the Code Cache’s Control-Flow.

4.4.5 Linking Updates

As we have seen previously, Pin is performing linking operations between different Code Cache Traces, in order to reduce the Code Cache accesses. For this reason, our Pintool creates a Link Table that contains information about the new transitions, and exactly when they should be applied to the Control Flow Table. Pintool correlates every discovered link to the next Code Cache Trace, that the execution of Pin has visited, and stores them inside a Shared Memory Hash table. This Link Table has been created, in order to allow the Decoder to be able to know when the links should be applied, during the reconstruction algorithm. Therefore, each time the Decoder starts the Control-Flow reconstruction of a specific Code-Cache, all the links that correspond to it need to be performed. For this reason, when a Code Cache Entry TIP appears, and after the Code Cache address is discovered, our Decoder accesses the Hash Table using the address of the Trace. All the new transitions, due to linking, for that specific Code Cache Trace exist at the accessed Hash table cell. Using those transitions, the Decoder is able to modify the proper entries of the

Control Flow Table, and thus insert the new links. This way, the Control Flow Table is updated every time a new link between two Caches appears, and the Control-Flow can be correctly reconstructed by our algorithm.

4.4.6 Control-Flow Reconstruction

Our work focuses on reconstructing the Control-Flow of an application executed inside a JIT environment. Therefore, the reconstruction algorithm is one of the most important components of our system. The reconstruction algorithm utilizes the collected Intel PT packets and the Control-Flow Table, in order to extract all the Basic Blocks of the original execute application.

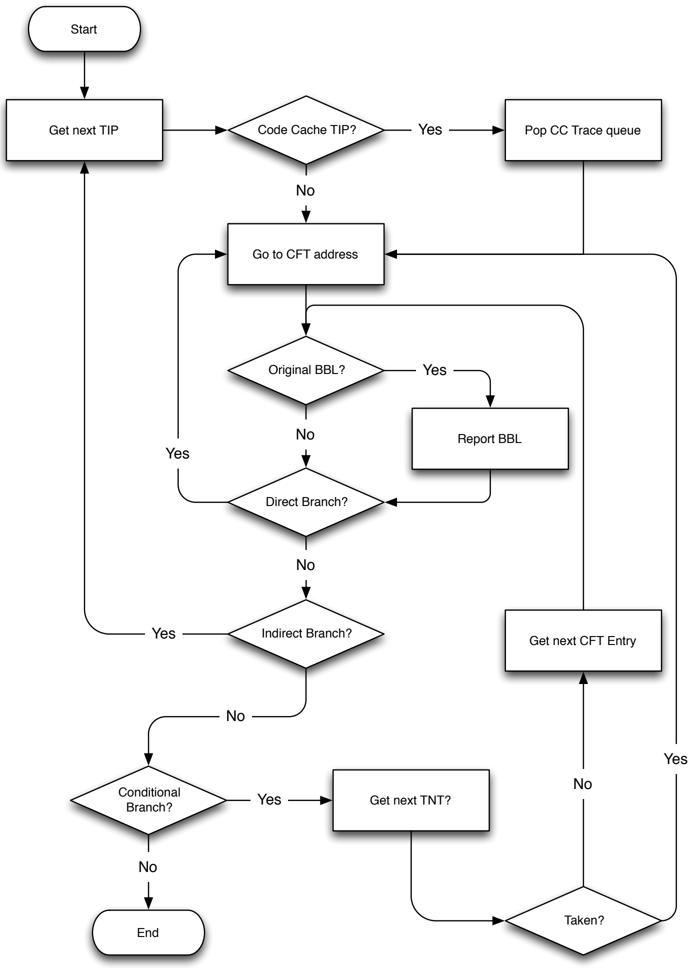


Figure 4.4: Flow-Chart of the Reconstruction Algorithm.

Figure 4.4 depicts the Flow-Chart of our reconstruction algorithm. Our algorithm is based on reconstructing the Control-Flow of each Code Cache Trace, that Pintool accessed during it's execution. Each time a Code Cache Entry TIP is decoded the reconstruction begins, since this packet indicates that all the subsequent decoded packets will refer to the execution of a Code Cache Trace. We discover the address of the specific Code Cache Trace by getting the first address of the previously described Trace queue. The algorithm then accesses the Control-Flow Table at the position where the Entries of that specific Code Cache Trace are placed and begins to follow the same execution that Pintool itself did. This is done by using each Entry of the table in order to figure out the next Entry that should be accessed. If one the accessed Entries contains an Original Basic Block Address, it means that it is a Basic Block of the original application and it is reported to the system. If an Entry is just an Original Basic Block Address and doesn't contain a transition, the algorithm continues to the next Entry of the table. Each time a transition is met, it is analyzed in order to find out whether Intel PT information are required to resolve it's Target or not. The direct branches are simply followed by using their Target address in order to move inside the Control-Flow Table. In case where an indirect or conditional branch is met, Intel PT packets are required to resolve the transition. If an indirect branch is found, the algorithm uses the next TIP packet in order to find out the destination of this transition. This address, acquired from the decoded TIP packet, is used to move at this specific location inside the Control-Flow Table. If a conditional branch is found, the next TNT packet is used, to discover whether this branch is taken or not. If the next decoded TNT packet contains the Taken bit, the Target of the conditional branch entry is used to move inside the Control-Flow Table and continue the reconstruction from there. If the bit of the next TNT packet is Not Taken, the execution trailing continues to the next Entry of the Control Flow Table, since this is the case of a fall-through instruction.

Following this procedure for every Code Cache Trace, results in correctly following the execution of Pintool inside the Code Cache and extracting the Basic Block of the original application. Subsequently, this leaves us with the complete reconstructed Control-Flow of the application, that is executing inside the JIT environment.

Chapter 5

Evaluation

In this chapter we discuss the evaluation of our system. We have tested our implementation on a Intel Xeon E3-1225 CPU with 8GB of DDR4 RAM. The SPEC CPU2006 benchmark suite has been used in order to evaluate the correctness and performance of the system.

5.1 Correctness

Our system focuses on discovering the execution of an original program executed inside a JIT environment and reconstructing its Control-Flow. In order to test that the complete system is able to correctly reconstruct the execution flow of a JITed application, the actual Control-Flow of the program is required. As Control-Flow, we refer to the Basic Blocks that the program visited through its execution. By comparing the actual Basic Blocks to the ones collected by our system, we can ensure that our implementation can correctly perform the reconstruction of the application.

We have utilized Pin and its API to collect the complete Control-Flow of the application. More specifically, we have used the Instrumentation API of Pin to extract all the Basic Blocks of the original application. This is performed by adding an instrumentation call at the first address of each original Basic Block, using INS_InsertCall. This way we are able to log the address of the first instruction for each Basic Block of the original application. We consider this as the ground-truth to our systems evaluation. Comparing the Basic Blocks collected through implementation to the ones acquired by our reconstruction tool, we are able to ensure the correctness of our system.

In order to ensure that the correctness evaluation is performed accurately, we have used our systems Pintool to collect the Basic Block through instrumentation. Having the instrumentation code inside the same Pintool, means that our evaluation is not affected by ASLR, since both the Control-Flows are collected at the same execution of the application. By simply comparing the collected Basic Blocks, we are able to ensure that the reconstruction was performed correctly.

The SPECint benchmarks, that come with SPEC CPU2006 benchmark suite, have been used during our evaluation. By default, the SPECint benchmark comes with runspec, a tool that is responsible for building, running and reporting the benchmarks. Since we only care about reconstructing the Control-Flow of the benchmarks, the runspec script is not required. We have manually compiled the benchmarks using gcc and executed them through Pintool. We compared the collected Control-Flows using a python script. The results indicate that the collected Control-Flow by our system is identical to the one acquired by instrumenting each Basic Block of the original application. This means that, our system is able, by utilizing Intel PT, to correctly trace and reconstruct the Control-Flow of an application executing inside a JIT environment.

5.2 Performance

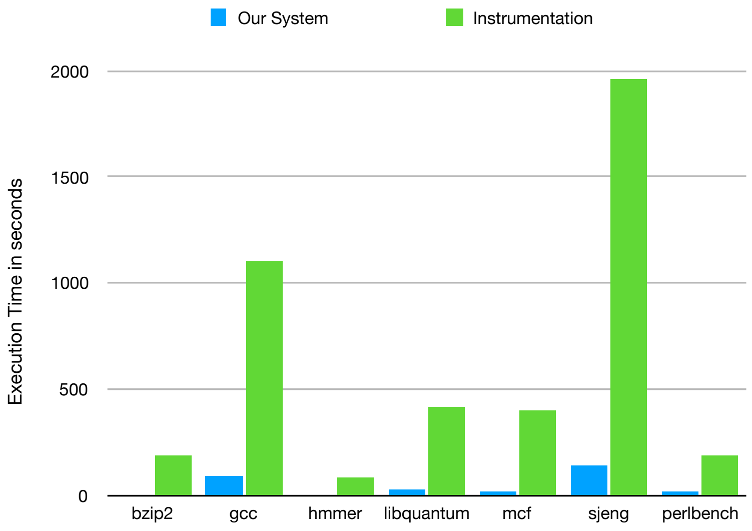


Figure 5.1: The execution time of our approach compared to this of the instrumentation technique.

In order to evaluate the performance of our mechanism, we used a subset of SPEC2006 benchmark suite, and we compared its execution times to a Pintool that obtains the Control-Flow using instrumentation per Basic Block. Figure 5.1 displays the execution runtime of our System compared to the instrumentation one. We can easily notice that our system is able to reconstruct the Control-Flow of the application marginally faster than the approach offered by Intel. Another interesting finding, that is indicated on Figure 5.2, is that the reported overheads are based on the number of the Basic Blocks of each benchmark.

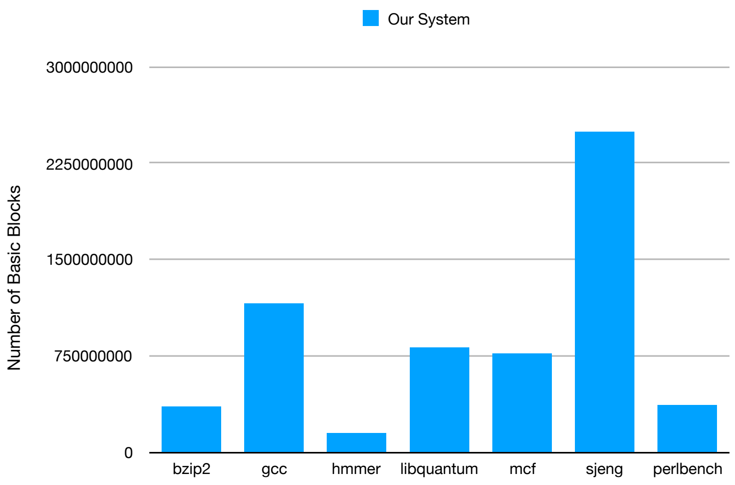


Figure 5.2: The number of Basic Blocks for each benchmark.

Chapter 6

Related Work

Intel Processor Trace is a new hardware feature recently introduced by Intel, mainly for debugging purposes. Even though the offered functionality can be quite useful to systems requiring to extract the Control-Flow of an application, it hasn't been fully utilized yet. There are certain works that have used Intel PT in order to trace the execution of an application, to either the user or the kernel space. However, none of them is focusing on tracing a program that is executing inside a Just-in-Time environment. In this section, we present other works that have utilized this hardware feature in order to trace the execution of an application and reconstruct its Control-Flow.

6.1 kAFL

KAFL [33] is a kernel fuzzer that utilizes the Intel Processor Trace. They introduced a new technique that allows applying feedback fuzzing to arbitrary (even closed source) x86-64 based kernels, without any custom ring 0 target code or even OS-specific code at all. Intel PT provides Control-Flow information on running code. They use this information to construct a feedback mechanism similar to AFL's instrumentation. Their system is able, by utilizing the Intel PT data and the disassemble of the binary, to extract the Basic Blocks that the execution has followed. However, their approach is only limited on tracing the Kernel side of the application.

6.2 PT-CFI

PT-CFI [34] is a backward-edge CFI model based on Intel PT. More specifically, they have implemented a new practical CFI model for native COTS binaries based on the traces from Intel PT. However, their approach doesn't include reconstructing the complete Control-Flow of the application and comparing it to the CFG, since as they claim this would be quite costly. Instead, the collected traces are used in order to build a TIP sequence graph, and compare this graph with the legitimate TIP graph.

6.3 GRIFFIN

GRIFFIN [35] is a hardware-assisted, operating system CFI enforcement mechanism, that uses Intel PT to protect user-space processes from Control-Flow hijacking attacks. It is capable of enforcing stateful CFI policies on both forward and backward edges and is designed to leverage Intel PT traces efficiently. When enforcing coarse-grained CFI policies, it uses TIP packets in the trace directly. GRIFFIN reconstructs the complete Control-Flow from Intel PT traces in parallel to enforce fine-grained CFI policies. Even though their approach includes reconstructing the complete Control-Flow of an application or JIT code, it doesn't cover the discovery of the Basic Blocks of the actual program that is executing inside the JIT environment.

6.4 POMP

POMP [36] is an automated tool to facilitate the analysis of post-crash artifacts. More specifically, POMP introduces a new reverse execution mechanism to construct the data flow that a program followed prior to its crash. It utilizes Intel PT in order to perform a backward tracing of the execution, starting from the point where the crash occurred. It then performs taint analysis and highlights those program statements that actually contribute to the crash. POMP is able to reconstruct the Control-Flow of an application in a backward manner, until it reaches the point where the crash originated from. However, it doesn't offer long-time tracing of the application, as it only holds the latest Intel PT data.

6.5 FlowGuard

FlowGuard [37] uses Intel PT to efficiently collect runtime Control-Flow traces, which are compared with a statically derived Control-Flow graph to detect abnormalities. FlowGuard constructs a Control-Flow graph offline, using static binary analysis. This flow graph is in a form that can be directly compared with Intel PT traces. FlowGuard adopts a hybrid flow checking mechanism that separates the fast and slow paths. Most of the runtime flow traces can be checked in a fast path by directly searching on the reconstructed CFG, while only some rare abnormal flow traces are passed to the slow path for precise CFI checking and enforcement. The complete Control-Flow is reconstructed when the slow path is used. Their approach is adaptive to the occasion, as it can switch between comparing the Intel PT trace data directly to the Control-Flow graph or reconstruct the complete Control-Flow of an application.

6.6 ProRace

ProRace [38] proposes a new methodology to reconstruct unsampled memory addresses using the Control-Flow trace collected at runtime. In order to recover more memory accesses around each sample, ProRace collects the Intel PT traces at runtime. The Intel

PT data are then used as a guide, in order to decide which path to take during the offline replay. This enables ProRace to reproduce many other unsampled memory operations preceding and following each sample along the observed program path. ProRace is able to reconstruct the complete Control-Flow of an application, but their approach only requires this to be performed offline and not during the execution of the program.

6.7 Gist

Gist [39] is a failure sketching tool, that provides developers with an explanation of the root cause of a failure that occurred in production. Gist performs data and Control-Flow tracking in a cooperative setup by targeting either multiple executions of the same program in a data center or users who execute the same program. It uses hardware watchpoints to track the values of data items, and Intel PT to trace the Control-Flow. Gist combines static program analysis and the tracing data to produce the failure sketches. By using static analysis, Gist determines the locations where Control-Flow tracking should be performed, meaning that only the Control-Flow of certain areas of interest is reconstructed. Since the failure sketching is performed offline, Gist doesn't perform real-time Control-Flow reconstruction of the application.

6.8 INSPECTOR

INSPECTOR [40] is a data provenance library for multi-threaded programs. Their approach targets existing executables and relies on OS-specific mechanisms and Intel PT to efficiently build the Concurrent Provenance Graph (CPG). The CPG records control, data, and schedule dependencies for the shared-memory multi-threaded program execution. INSPECTOR uses the Intel PT data to create the CPG, but it doesn't reconstruct the complete Control-Flow of the application.

Chapter 7

Future Work

At this moment our system does not support multi-threaded processes, since reconstructing the Control-Flow requires the use of additional packets. Multi-threaded applications produce Intel PT packets that allow the Decoder to identify each thread and trace its execution separately to the others. Our plan is to utilize these Intel PT packets and additional Kernel Ring buffers, in order to fully support the tracing and Control-Flow reconstruction of multi-threaded processes.

Our tool suffers from certain slowdowns in cases where the Intel PT data are produced faster that consumed. Several optimizations can be performed in order to avoid slowdowns. These include introducing larger and more sub-buffers or even a separate Ring buffer that can hold data that would otherwise be overwritten by the hardware. In addition, copying the data from Kernel to User space can be considered as an option in some cases, so that the Decoder can analyze the data without the danger of them being overwritten. This would allow the main process to continue with its execution, instead of being blocked, even if the Intel PT packets have not yet been used to reconstruct the Control-Flow.

Furthermore, another aspect that we would like to explore is whether our tool can be adopted to other JIT environments. Our system is able to trace the execution of a process that is being executed by Pin. By creating a mapping of the code being generated between different JIT environments, we can enable our system to expand its support to other JIT engines.

Finally, the next step of this project is to utilize the reconstructed Control-Flow of the application in order to execute the DFT algorithm. Our main idea is to use the discovered Basic Blocks of the process during runtime in order to execute DFT in parallel. This will enable us to decouple analysis from execution and utilize spare CPU cores in order to avoid slowdown to the main process.

Chapter 8

Conclusion

Control-Flow reconstruction has been utilized by many security, profiling and analysis mechanisms throughout the years. Existing mechanisms of obtaining the Control-Flow of a process, based on instrumentation, suffer from severe slowdowns and inaccuracies. Recent works have shown that it is possible to obtain the Control-Flow of a process in a reliable and fast manner, by utilizing the hardware in order to trace the execution. However, none of these approaches, to the best of our knowledge, focuses on tracing the execution of an application that is executed inside a JIT environment. Until now, the Control-Flow of a JITed application could only be acquired by instrumenting the Basic Blocks of the original application. This technique imposes huge slowdowns and cannot be utilized for enforcing run-time protection mechanisms, such as CFI and DFT.

In this work we aim to provide an efficient and accurate way of tracing an application that is executing inside a JIT environment, while imposing significantly less overhead than the instrumentation approaches. We leverage Intel Processor Trace, a new hardware feature of modern Intel CPUs, to acquire the Control-Flow of a JITed process correctly, while at the same time minimizing the impact on the performance. In order to showcase our mechanism, we trace the execution of a process inside Intel Pin dynamic instrumentation framework. In addition, we utilize certain API options of Pin itself to acquire information about the JITed code and create a table containing Control-Flow transitions inside the Code Cache. Our approach does not include any instrumentation calls, thus avoiding additional overhead at the Pin side. By utilizing the recorded traces by Intel PT and the Control-Flow table created by Pin itself, our system is able to isolate the execution of the original code from the JIT code and reconstruct its complete Control-Flow. Our prototype is based on a custom Intel PT Driver, Intel Pin and an Intel PT Decoder. We evaluate the whole system by running the SPEC2006 benchmark suit. Our evaluation proves that our system is able to correctly isolate the original code of a process from inside the JIT code and accurately reconstruct its Control-Flow during runtime. The imposed overhead is marginally lower than instrumentation approaches, while at the same retaining the same accuracy levels.

Bibliography

[1] J. Newsome and D. Song, "Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software," 2005.

[2] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham, "Vigilante: End-to-end containment of internet worms," in *ACM SIGOPS Operating Systems Review*, vol. 39, no. 5. ACM, 2005, pp. 133–147.

[3] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena, "Bitblaze: A new approach to computer security via binary analysis," in *International Conference on Information Systems Security*. Springer, 2008, pp. 1–25.

[4] D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz, "Bap: A binary analysis platform," in *International Conference on Computer Aided Verification*. Springer, 2011, pp. 463–469.

[5] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti, "Control-flow integrity principles, implementations, and applications," *ACM Transactions on Information and System Security (TISSEC)*, vol. 13, no. 1, p. 4, 2009.

[6] G. Ammons, T. Ball, and J. R. Larus, "Exploiting hardware performance counters with flow and context sensitive profiling," *ACM Sigplan Notices*, vol. 32, no. 5, pp. 85–96, 1997.

[7] T. Ball and J. R. Larus, "Efficient path profiling," in *Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture*. IEEE Computer Society, 1996, pp. 46–57.

[8] T. M. Chilimbi, B. Liblit, K. Mehra, A. V. Nori, and K. Vaswani, "Holmes: Effective statistical debugging via efficient path profiling," in *Proceedings of the 31st International Conference on Software Engineering*. IEEE Computer Society, 2009, pp. 34–44.

[9] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," in *ACM Sigplan notices*, vol. 42, no. 6. ACM, 2007, pp. 89–100.

[10] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, “Control-flow integrity,” *Proceedings of the 12th ACM conference on Computer and Communications Security (CCS)*, 2005.

[11] r. roemer, e. buchanan, h. shacham, and s. savage, “Return-Oriented Programming: systems, languages, and applications,” *Proceeding of ACM Transactions on Information and System Security (tissec)*, 2012.

[12] J. Li, Z. Wang, X. Jiang, M. Grace, and S. Bahram, “Defeating return-oriented rootkits with "Return-Less" kernels,” *Proceedings of the 5th European conference on Computer systems (EuroSys)*, 2010.

[13] R. Hund, T. Holz, and F. C. Freiling, “Return-oriented rootkits: Bypassing kernel code integrity protection mechanisms,” in *USENIX Security Symposium*, 2009, pp. 383–398.

[14] Z. Wang and X. Jiang, “Hypersafe: A lightweight approach to provide lifetime hypervisor control-flow integrity,” in *Security and Privacy (SP), 2010 IEEE Symposium on*. IEEE, 2010, pp. 380–395.

[15] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou, “Practical control flow integrity and randomization for binary executables,” in *Security and Privacy (SP), 2013 IEEE Symposium on*. IEEE, 2013, pp. 559–573.

[16] M. Payer, A. Barresi, and T. R. Gross, “Fine-grained control-flow integrity through binary hardening,” in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2015, pp. 144–164.

[17] L. Davi, A.-R. Sadeghi, and M. Winandy, “Ropdefender: A detection tool to defend against return-oriented programming attacks,” in *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*. ACM, 2011, pp. 40–51.

[18] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda, “Panorama: capturing system-wide information flow for malware detection and analysis,” in *Proceedings of the 14th ACM conference on Computer and communications security*. ACM, 2007, pp. 116–127.

[19] K. H. Lee, X. Zhang, and D. Xu, “High accuracy attack provenance via binary-based execution partition.” in *NDSS*, 2013.

[20] V. Chipounov, V. Kuznetsov, and G. Candea, “S2e: A platform for in-vivo multi-path analysis of software systems,” *ACM SIGPLAN Notices*, vol. 46, no. 3, pp. 265–278, 2011.

[21] A. Slowinska, T. Stancescu, and H. Bos, “Howard: A dynamic excavator for reverse engineering data structures.” in *NDSS*, 2011.

[22] D. Y. Zhu, J. Jung, D. Song, T. Kohno, and D. Wetherall, "Tainteraser: Protecting sensitive data leaks using application-level taint tracking," *ACM SIGOPS Operating Systems Review*, vol. 45, no. 1, pp. 142–154, 2011.

[23] K. Jee, V. P. Kemerlis, A. D. Keromytis, and G. Portokalidis, "Shadowreplica: efficient parallelization of dynamic data flow tracking," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM, 2013, pp. 235–246.

[24] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," in *Acm sigplan notices*, vol. 40, no. 6. ACM, 2005, pp. 190–200.

[25] D. Bruening, Q. Zhao, and S. Amarasinghe, "Transparent dynamic instrumentation," *ACM SIGPLAN Notices*, vol. 47, no. 7, pp. 133–144, 2012.

[26] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity," in *Proceedings of the 12th ACM conference on Computer and communications security*. ACM, 2005, pp. 340–353.

[27] N. Oh, P. P. Shirvani, and E. J. McCluskey, "Control-flow checking by software signatures," *IEEE transactions on Reliability*, vol. 51, no. 1, pp. 111–122, 2002.

[28] D. Bruschi, L. Martignoni, and M. Monga, "Detecting self-mutating malware using control-flow graph matching," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2006, pp. 129–143.

[29] Intel, "libipt - Intel Processor Trace Decoder Library." https://github.com/01org/processor-trace.

[30] ——, "Pin 3.2 User Guide." http://people.cs.uchicago.edu/~xiliang.

[31] ——, "XED User Guide." https://software.intel.com/sites/landingpage/pintool/docs/65163/Xed/html.

[32] A. Kleen, "simple-pt - Simple Intel CPU processor tracing on Linux." https://github.com/andikleen/simple-pt.

[33] S. Schumilo, C. Aschermann, R. Gawlik, S. Schinzel, and T. Holz, "kAFL: Hardware-Assisted Feedback Fuzzing for OS Kernels." *Proceedings of the 26th Security Symposium (USENIX Security)*, 2017.

[34] Y. Gu, Q. Zhao, Y. Zhang, and Z. Lin, "PT-CFI: Transparent Backward-Edge Control Flow Violation Detection Using Intel Processor Trace." *Proceedings of the 7th ACM International Conference on Data and Application Security and Privacy (CODASPY)*, 2017.

[35] X. Ge, W. Cui, and T. Jaeger, "GRIFFIN: Guarding Control Flows Using Intel Processor Trace." *Proceedings of the 22nd ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2017.

[36] J. Xu, D. Mu, X. Xing, P. Liu, P. Chen, and B. Mao, "(POMP: Postmortem Program Analysis with Hardware-Enhanced Post-Crash Artifacts," *Proceedings of the 26th Security Symposium (USENIX Security)*, 2017.

[37] Y. Liu, P. Shi, X. Wang, H. Chen, B. Zang, and H. Guan, "Transparent and Efficient CFI Enforcement with Intel Processor Trace." *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2017.

[38] T. Zhang, C. Jung, and D. Lee, "ProRace: Practical Data Race Detection for Production Use." *Proceedings of the 22nd ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2017.

[39] B. Kasikci, B. Schubert, C. Pereira, G. Pokam, and G. Candea, "Failure sketching: A technique for automated root cause diagnosis of in-production failures." *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP)*, 2015.

[40] J. Thalheim, P. Bhatotia, and C. Fetzer, "INSPECTOR: Data Provenance using Intel Processor Trace (PT)," *Proceedings of the 36th International Conference on Distributed Computing Systems (ICDCS)*, 2016.