Thesis Dissertation

DEFENDING RETURN ORIENTED PROGRAMMING ATTACKS USING INTEL PROCESSOR TRACE

Loizos Nicolaou

UNIVERSITY OF CYPRUS

COMPUTER SCIENCE DEPARTMENT

May 2023

UNIVERSITY OF CYPRUS
COMPUTER SCIENCE DEPARTMENT

Defending Return Oriented Programming Attacks Using Intel Processor Trace

Loizos Nicolaou

Supervisor
Dr. Elias Athanasopoulos

Thesis submitted in partial fulfilment of the requirements for the award of degree of Bachelor in Computer Science at University of Cyprus

May 2023

Acknowledgments

This paper is dedicated to my beloved parents Giota Leonidou Nicolaou and Nicos Nicolaou for their support throughout the years. Grateful acknowledgment to Dr. Elias Athanasopoulos for his supervision.

Summary

Return-Oriented Programming (ROP) is a complex method of exploiting vulnerabilities in software applications. It involves constructing a chain of small code sequences, known as gadgets, that can be reused to perform unintended operations within the application. However, existing methods for defending against ROP are limited. Some are limited to the depth of the analysis, while others require specialized compiler, which compromise the integrity of the application binary.

This thesis dissertation presents the implementation and evaluation of eavesdROP tool. EavesdROP is our approach for non-branch limited attack detection and prevention of Ret-type Return Oriented Programming(ROP) attacks. It is built on top of Intel Processor Trace, a hardware feature that provides low overhead control flow tracing of a process. EavesdROP, designed for Linux machines makes use of ptrace facility along with Intel PT feature mention before and Libipt, Intel's reference implementation library for decoding Intel PT traces. EavesdROP is able to perform dynamic, transparent, variable depth, Call-Ret imbalance heuristic analysis of a statically compiled target application, in pursuance to determine whether it is undergoing a ROP attack.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

In C, memory management is the responsibility of the programmer. This means that it is up to the programmer to correctly allocate and deallocate memory in their C program. If memory is not managed properly, it can lead to a variety of problems, including buffer overflows.

A buffer overflow occurs when a program writes more data to a buffer (a temporary storage area in memory) than the buffer was designed to hold. This can cause the excess data to overwrite adjacent memory, potentially corrupting or overwriting important data. In some cases, a buffer overflow can be exploited by an attacker to execute arbitrary code, allowing them to take control of the program or system.

The most effective method of preventing buffer overflow attacks is through the implementation of secure code. However, additional countermeasures, including DEP/NX(Data Execution Prevention/No Execution)[1], ASLR(Address Space Layout Randomization)[25], and stack canaries[8], have also been introduced to protect against these types of exploits.

In this dissertation, inspired by kBouncer[18]: Efficient and Transparent ROP Mitigation, research will be conducted to the detection of a specific exploit technique, ROP(Return Oriented Programming) used to bypass DEP memory protection feature, using Intel PT(Intel Processor Trace) a CPU tracing feature which records the program execution.

In order to achieve this, numerous techniques are used to observe and control dynamically the execution of the target application, in pursuance to analyse its behaviour and determine whether its undergoing an attack.

Chapter 2

Background

Buffer Overflow

A buffer overflow is a type of vulnerability in a computer program that occurs when the program tries to store more data in a buffer than it was designed to hold. This can cause the program to crash, or it can allow an attacker to execute malicious code.

In the C programming language, buffer overflows can occur when a program tries to store data in an array or string that is too large for the allocated buffer. This can happen when the program does not properly check the size of the input data before trying to store it.

Buffer overflows can be exploited by attackers to execute arbitrary code, allowing them to gain unauthorized access to a system or to perform other malicious actions. They are a common type of security vulnerability and are often found in programs written in C or C++.

DEP/NX

Data Execution Prevention (DEP) is a system-level memory protection security feature that is designed to prevent malicious code from being executed in memory regions or certain pages that are not intended for execution. DEP is implemented in hardware and software, and it works by marking certain areas of memory as non-executable. This means that if an attacker tries to execute code from these areas, DEP will block the execution and prevent the attack from taking place.

NX (No eXecute) is a technology that is similar to DEP. It is implemented in hardware and allows the operating system to mark certain areas of memory as non-executable. When NX is enabled, any attempt to execute code from a non-executable memory area will result in an exception being raised.

Both DEP and NX are designed to protect against buffer overflow attacks and other types of attacks that involve executing malicious code in memory. They are commonly used in modern operating systems and are an important part of the security landscape.

Address Space Layout Randomization(ALSR)

Address Space Layout Randomization (ASLR) is a security technique that helps to protect computer systems from exploitation by randomizing the memory addresses of various system components, such as executable code, libraries, and data. The goal of ASLR is to make it more difficult for attackers to predict the memory addresses of vulnerable system components and thus prevent them from launching successful attacks.

ASLR is especially effective against Return-Oriented Programming (ROP) attacks, which are a type of memory corruption attack that involves manipulating the return address of a function to redirect program execution to arbitrary locations in memory, where an attacker can execute malicious code. By randomizing the memory addresses of system components, ASLR makes it much more difficult for attackers to determine the correct memory addresses to use in their ROP chains, which in turn makes it more difficult for them to exploit vulnerabilities.

However, ASLR can be bypassed in certain circumstances. One common technique used to bypass ASLR is to perform memory disclosure attacks, which involve exploiting a vulnerability to leak the memory address of a system component that is not randomized by ASLR, such as a shared library. Once an attacker has the address of one component, they can use it to calculate the addresses of other components and launch their ROP attack.

Another technique used to bypass ASLR is to use a brute-force attack, where an attacker attempts to guess the correct memory address by repeatedly trying different addresses until they find the correct one. This technique is generally only effective against poorly implemented ASLR or when combined with other vulnerabilities or techniques.

Control Flow Integrity (CFI) checks

Control Flow Integrity (CFI) is a security technique designed to defend against code reuse attacks such as Return-Oriented Programming (ROP) attacks. CFI works by enforcing constraints on the control flow of a program, such that the program can only execute instructions in a valid order according to its control flow graph. This is accomplished by adding metadata to the binary code of the program, such as information about the valid targets of a particular function call. At runtime, the CFI system checks the metadata to ensure that the program is following a valid control flow path or not.

However, there are several drawbacks to CFI as a defense mechanism. One of the primary drawbacks is the performance overhead of CFI, as it requires additional runtime checks and metadata processing. Additionally, CFI can be bypassed by attackers who are able to control the flow of the program in ways that are still considered valid by the CFI system. For example, some ROP attacks use unaligned gadgets, which are sequences of instructions that do not start at the beginning of an instruction boundary, and therefore do not match the expected control flow path. These types of attacks can be difficult for CFI systems to detect, as they may be interpreted as legitimate control flow changes by the system.

Dynamic Binary instrumentation

Instrumentation-based approaches are a class of security techniques designed to defend against code reuse attacks such as Return-Oriented Programming (ROP). These techniques work by inserting additional code, known as "check code," into the binary code of a program that checks for violations of the expected control flow. At runtime, the check code monitors the execution of the program and verifies that the program is following a valid control flow path. If the check code detects a deviation from the expected control flow, it can terminate the program or take other protective measures to prevent further damage.

One drawback of instrumentation-based approaches is the overhead they impose on program execution. Because the check code must execute alongside the program code, it can slow down the performance of the program and increase its memory footprint. Additionally, attackers can attempt to evade the check code by manipulating the program in ways that do not trigger the expected checks. For example, attackers may modify the code in such a way that the check code is not executed, or they may attempt to bypass the checks by exploiting weaknesses in the instrumentation mechanism itself.

Furthermore, attackers can use polymorphism and other evasion techniques to modify the program's control flow in ways that bypass the check code. Polymorphism involves modifying the program code at runtime to generate new variations of the attack code, which can be difficult for the check code to detect. Additionally, attackers can use obfuscation techniques to make the code harder to read and analyze, making it more difficult for the check code to identify deviations from the expected control flow.

Return-oriented programming (ROP)

Is a technique used by attackers to bypass data execution prevention (DEP) and other memory protection measures. It involves chaining together short segments of code called "gadgets" that are already present in a program's memory, in order to execute arbitrary code.

In ROP, the attacker does not inject new code into the program's memory. Instead, they manipulate the program's execution flow to execute existing code in a way that was not intended by the original developer. This can allow the attacker to bypass DEP and other memory protection measures, since the code being executed is already present in the program's memory and was not introduced by the attacker.

ROP is often used in conjunction with other techniques, such as buffer overflow attacks, to compromise the security of a system.

Jump Oriented Programming(JOB) - similar to ROP, but instead of reusing gadgets, an attacker uses a series of indirect jumps to execute malicious code.

Call Oriented Programming(COP) - an advanced form of ROP where the attacker uses the existing code to build a chain of function calls that execute the malicious code.

PID

A PID, or process ID, is a unique numerical identifier assigned to each process running on a computer. It is used to identify and track individual processes, and is typically assigned by the operating system when a new process is created.

The PID is a unique positive integer value, and it's used by the operating system to keep track of the process, manage its resources, and control its execution. For example, the operating system uses the PID to locate the process's memory space, open files, and system resources, and to send signals to the process.

SIGSTOP

SIGSTOP is a signal in the Unix operating system that is used to stop the execution of a process. When a process receives a SIGSTOP signal, it will immediately stop executing and will not be able to continue until it receives a SIGCONT signal.

SIGSTOP is a "non-catchable, non-ignorable" signal, which means that the process cannot catch or ignore it using a signal handler. This makes it a useful tool for forcefully stopping a process that may be stuck in an infinite loop or otherwise unresponsive.

Intel PT

Intel Processor Trace (Intel PT) is a hardware feature of certain Intel processors that allows for the tracing and recording of the execution of instructions on the processor. It can be used for a variety of purposes, such as debugging and performance analysis.

Intel PT works by constantly recording the flow of instructions as they are executed by the processor. The recorded data can be accessed later for analysis, providing a detailed record of the processor's execution. Intel PT can be used to trace the execution of code at the instruction level, allowing for a more fine-grained analysis of the processor's behavior.

Intel PT Type Intel PT type is a numerical value located in

```
/sys/bus/event_source/devices/intel_pt/type
```

The "type" file within this subdirectory provides information about the specific type of Intel PT feature that is present on the system. This information can be useful for determining the capabilities and supported protocols of the Intel PT feature, as well as for identifying the generation of the CPU that the system is running on.

File descriptor

File descriptor is an abstract indicator used to access a file or other input/output resources, and it is a non-negative integer, it's unique within a process. When a file is opened, the operating system returns a file descriptor that can be used to read and write to the file. In the case of mmap, the file descriptor is used to specify the file that is being mapped into memory.

Libipt

The Intel Processor Trace (Intel PT) Decoder Library is Intel's reference implementation for decoding Intel PT.

The Libipt library provides an API for working with Intel PT data, allowing developers to write tools and applications that can make use of this feature.

XED

Intel XED is a software library developed by Intel that allows programmers to work with the X86 instruction set, which is used by most computers that run on the x86 architecture. It provides functions for encoding and decoding instructions, as well as tools for disassembling and assembling code.

Perf Events

Perf Events, also known as "Performance Events" or "perf," is a Linux kernel subsystem that provides a framework for collecting and analyzing performance data from the operating system and running applications.

It allows a process to monitor various events, such as CPU instructions executed, cache misses, page faults, and context switches, and collect data about these events in real-time or offline.

Perf Events provides a range of features, including the ability to:

- Collect data from multiple CPUs and processors simultaneously
- Sample data at a specified rate or in response to specific events
- Filter data based on process, thread, or CPU
- Profile the kernel, user space programs, or both

Ptrace

Ptrace is a system call that allows a process to be traced by another process. This means that the process being traced can be controlled and monitored by the tracer process. It can be used to monitor and control the execution of another process, and is often used for debugging and analyzing the behavior of programs. Some examples of what ptrace can be used for include examining the system calls that a process makes, injecting code into a process, and modifying the memory of a process.

waitpid

Waitpid is a system call in the Unix operating system that allows a parent process to wait for a specific child process to change state. By passing the process ID of the child process and options that control the behavior of the call, it allows the parent process to wait for the child process to change state and obtain the child's exit status, or wait for other specific state changes and handle them accordingly.

ioctl

ioctl (short for "input/output control") is a system call in Linux that allows a process to request input/output operations on a device or file. It is a general-purpose interface that can be used to perform various operations, such as reading or setting device parameters, initiating data transfer, or requesting device information.

The ioctl system call takes three arguments: a file descriptor, a request code, and an argument. The request code specifies the operation to be performed, and the argument points to a data structure or buffer that is used to pass additional information to the system call. The ioctl system call returns a positive value on success and a negative value on error.

Mmap

Mmap systemcall, also known as memory-mapped files,it allows a process to map a file or a portion of a file into its virtual memory address space. When a file is mapped, the process can read and write to the file directly using pointer operations, which can be more efficient than using the standard read and write system calls. The mmap function creates a new memory mapping for the specified file and the mapped region can be accessed as if it were an array in memory.

Statically compiled

Statically compiled refers to a method of building executable code from source code in which all of the necessary libraries and dependencies are included in the final executable file. In other words, when a program is statically compiled, all of the code required to run the program is contained within a single executable file, without the need for any external libraries or dependencies.

Statically compiled programs are often larger in size than dynamically compiled programs, as all of the libraries and dependencies are bundled with the program. However, they have the advantage of being self-contained, which means that they can be easily distributed and run on other systems without the need for any additional setup or configuration.

Statically compiled programs are also more resilient to changes in the system environment, as they do not rely on external libraries that may be updated or changed over time. This makes them a popular choice for certain types of software, such as system utilities or other low-level tools that need to run reliably across different environments.

Chapter 3

Architecture

3.1 eavesdROP approach

Our approach to detect ROP attacks is based on the implementation of runtime checks for abnormal control flow transfers. Conducting runtime checks for all control flow transfers can significantly increase the performance overhead of the system, as indirect control flow transfers are a common occurrence in executed code. To address this issue, we have proposed a refinement of the set of control transfers that need to be checked during runtime. This refinement is based on the observation that malicious code often relies on system calls to achieve its intended goals. Therefore, we propose that only the control transfers that occur within the final stages of the execution path leading to a system call should be subject to runtime checks. This reduction in the scope of control transfers that need to be checked can greatly reduce the performance overhead of the system while maintaining an appropriate level of security.

Abnormal control flow in ROP attacks is characterized by the manipulation of the program's execution flow through the use of gadgets as ilustrated in [3]. These gadgets are small code snippets that are already present in the program's memory, and are chained together by the attacker to redirect the program's execution flow to a location controlled by the attacker.

Before ROP code starts executing, the register that holds the stack pointer is set to the beginning of the ROP payload, this done through a stack pivot[10, 11] . Each gadget ends with a return instruction, which advances the stack pointer to the address of the next gadget, and transfers control to it. However, these return instructions of ROP code can be distinguished from legitimate return instructions of the actual program, since those are paired with call instructions(when one observes the instructions in order of execution). Moreover legitimate call instructions upon return tend to transfer execution control back to the next instruction from where the call was made. Therefore, an execution flow with dense return instructions which are not preceded by calls and lead to a system call, is considered abnormal behaviour and can be used as a marker for ROP code execution. This heuristic is later referred to as Call-Ret imbalance.

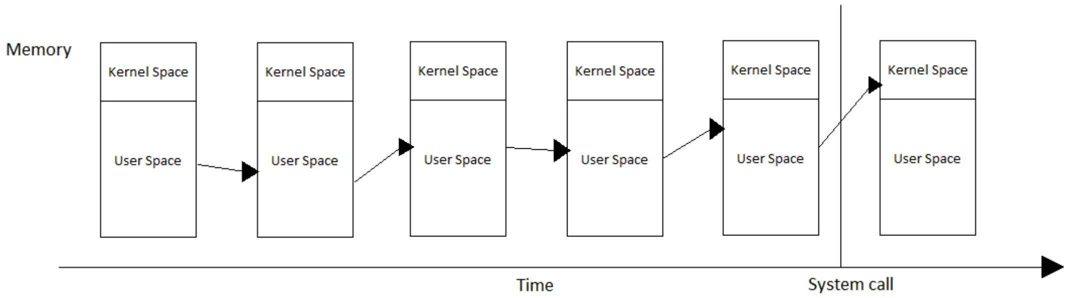

Figure 3.1: Upon kernel space entry, our monitoring program will check preceding indirect branches to determine whether arrived in that state through a benign system call or part of a ROP exploit [Visualization inspired by kBouncer[18]]

The image in Figure 3.1 illustrates the concept of memory snapshots and control transfers. The boxes represent snapshots of memory, and the arrows connecting them indicate the flow of control. The top section represents the kernel space and the bottom section represents the user space. The vertical line indicates the point at which control is transferred from user space to kernel space (usually through a system call such as syscall, sysenter, or int 0x80). This is the point where we examine the control flow path for any unusual transfers of control and determine if it is a legitimate system call or if it is being used as part of a ROP exploit.

We chose Intel PT a feature built into certain Intel processors that allows for the tracing and recording of a CPU's instruction execution. Intel PT benefits over the other approaches since its hardware based, meaning it does not dependent on software instrumentation and can trace instructions even in kernel or other privileged modes. It has minimal runtime overhead; it is fully transparent to the running processes; it can be dynamically enabled and requires no debugging symbols or source code.

Chapter 4

Implementation

4.1 Trace target application

In order to check preceding indirect branches to determine whether the subject process arrived in that state through a benign system call or as a part of a ROP exploit, we developed a tool that uses existing techniques to observe and control the execution of the target application, allowing us to perform runtime control transfer checks in a controlled manner.

To control the execution of the subject process, our tool uses ptrace, a system call primarily used to implement breakpoint debugging and system call tracing. Ptrace allows our tool, referred to from now on this chapter as the "Tracer", to attach and control another process, referred to as the "Tracee". Using this tracing feature, our tool can intercept and observe system calls made by the Tracee before they are executed. This allows us to analyse the preceding indirect branches the Tracee took to arrive there. Determining whether its under attack and therefore terminating its execution, or allowing it to continue until the next system call(or exit), in case of no malicious control transfer detection.

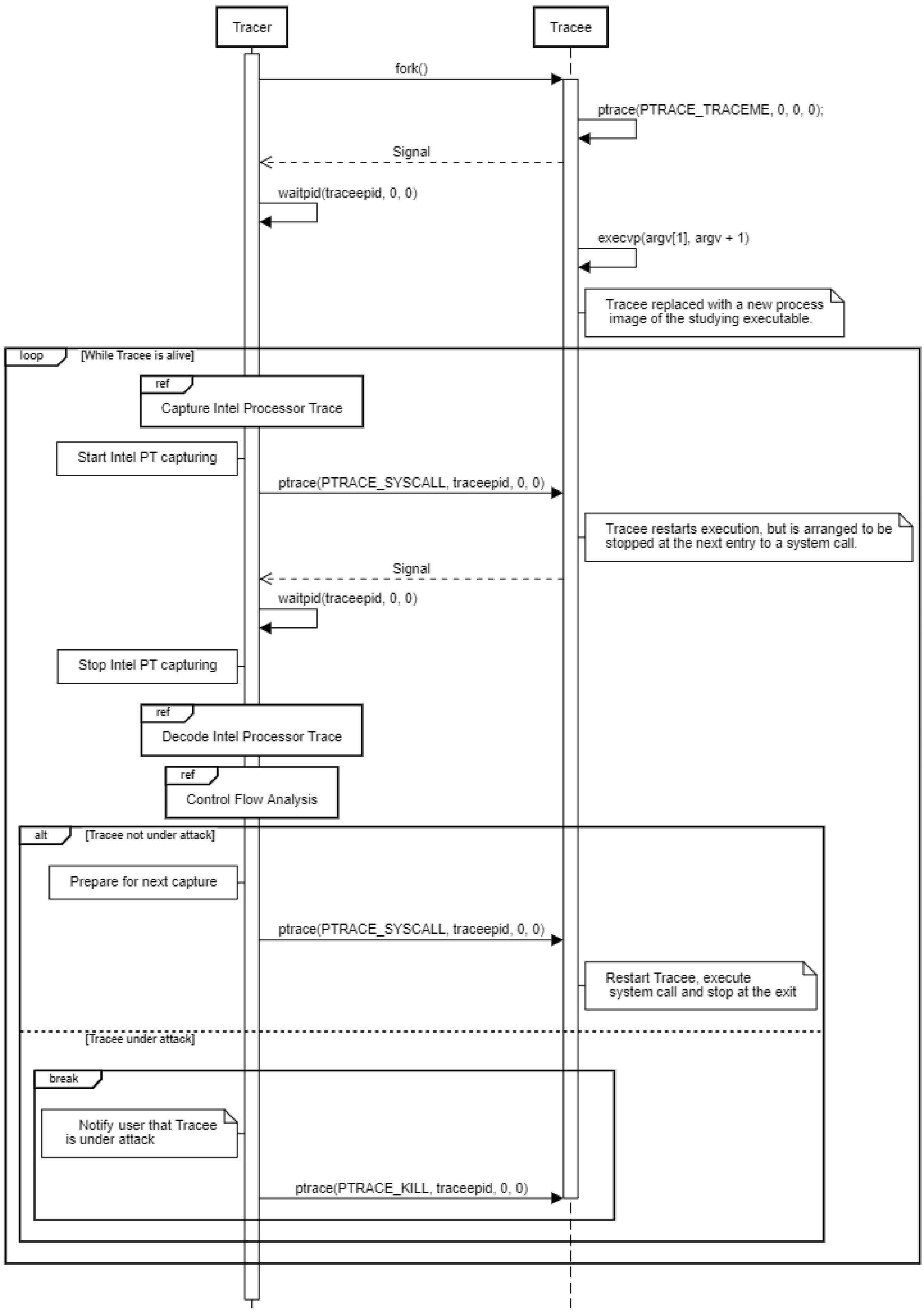

Figure 4.1: Unified Modeling Language (UML) diagram that illustrates the sequence of messages between our Tracer tool and the Tracee in an interaction.

The sequence diagram in Figure 4.1 illustrates the sequence of messages between our Tracer tool and the Tracee in an interaction. First the Tracer `forks()` to create a new process by duplicating itself. A `PTRACE_TRACEME` request is then initiated by the child, to turn it into a Tracee. This allows the child process to be controlled and traced by our tracer tool. In order to synchronize the Tracer with the Tracee a `SIGSTOP` signal is raised by the child. For this reason a `waitpid()` call is made by the Tracer, in order to detect this state change signal. The Tracee then continues by executing `execvp()` system call to replace its current process image with a new process image, the one of the target application we are interested on analysing. At this point the Tracer enters an infinite loop, to which, for every iteration, checks the preceding indirect branches leading to the system call the Tracee is about to take. This is achieved by initiating the Intel PT recording(discussed in more detail in the next chapter) and signaling the Tracee using `PTRACE_SYSCALL` request, to continue its execution, until the entry of the next system call. After execution the Tracee will halt upon system call entry and signal its stop to the Tracer. The Tracer who has been waiting for this state change signal, will terminate the Intel Pt recording, decode and analyse the control flow, in order to determine if it will allow the Tracee to continue execution. This procedure is repeated for all system calls the tracee makes and stops only if an attack is detected, or the target application finishes its execution. In case of a ROP attack detection the user is notified, the target application is terminated in order to avoid any malicious activities from taking place and the Intel Pt trace is automatically saved for further analysis.

4.2 Capture Intel Processor Trace

Intel Processor Trace (Intel PT) is an extension of Intel Architecture that can be used to trace software execution on Intel CPUs. It provides a record of software execution that can be used for debugging, profiling, and optimization purposes.This trace data is collected by a hardware unit called the Trace Hub, which is integrated into the CPU. The Trace Hub records the trace data in a buffer, and this data can then be retrieved and analyzed by software tools.

The trace data generated by Intel PT is in the form of packets, which are highly compressed binary representations of the information collected by the Trace Hub. Each packet contains a small amount of data, typically representing a single instruction, along with some additional metadata. The metadata includes information such as the type of packet, the size of the packet, and the address of the instruction being traced. This feature was first supported in Intel Core M and 5th generation Intel Core processors that are based on the Intel micro-architecture code name Broadwell. The information Intel PT provides allows for a more fine-grained analysis, providing a detailed record of the processor's execution and behaviour.

Perf Events, is a Linux kernel subsystem that provides a framework for collecting and analyzing performance data from the operating system and running applications. One of these performance events, is the Intel Processor Trace feature we chose to use in this research. The Perf Events tool has been available since Linux kernel version 2.6.31 in 2009 and comes with the `linux/perf_event` library, which allows for in-tool integration.

In order for our tool to record Intel PT[5] traces of the Tracee using `Perf_Events`, a file descriptor must be first obtained through which to talk to Perf subsystem. To do that we must first configure Perf Events according to our system. The `perf_event_attr` object holds all the configuration options Perf Events library requires for tracing. The structure is found in `/usr/include/linux/perf_event.h`. First we start by configuring the type of tracing we are performing(Intel PT). To do that we set the type option with the specific type of Intel PT feature that is present on the system. This type is used to determine the capabilities and supported protocols of the Intel PT feature, as well as for identifying the generation of the CPU the system is running. Since the type value differs between CPUs, our tool dynamically reads the host system's specific type numeric value from the Intel Pt type file located in the fixed path `/sys/bus/event_source/devices/intel_pt/type` of the Linux system. Furthermore other configuration flags are set to prepare tracing based on our needs, those are the `exclude_kernel` which tells the Perf Events subsystem not to count events that occur inside the kernel, such as interrupts or system calls, in the performance monitoring data. This is useful since our analysis is interested only in the performance of user-space code, and want to avoid the noise introduced by kernel activity.

Moreover the `start disabled` flag which tells Perf Events to start the tracing process disabled and finally the `precise_ip` option which is set to 3 indicating that we want no skid, meaning to record every instruction executed on the CPU. The corresponding code for the previously mentioned configuration is shown below.

```
struct perf_event_attr attr;
memset(&attr, 0, sizeof(attr));

attr.size = sizeof(attr);

FILE *pt_type_file = fopen("/sys/bus/event_source/devices/intel_pt/type", "r");
char pt_type_str[MAX_PT_TYPE_STR];
fgets(pt_type_str, sizeof(pt_type_str), pt_type_file)

attr.type = atoi(pt_type_str);

attr.exclude_kernel = 1;
attr.disabled = 1;
attr.precise_ip = 3; //No skid
```

After everything has been configured, we open the `perf_event` counter for Intel PT by calling `syscall(SYS_perf_event_open, &attr, traceePID, -1, -1, 0)` with the address of the configured `perf_event_attr` object, the process id of the Tracee, the -1 value to indicate measuring for all processes/threads on the specified CPU, 0 to show no group tracing and finally -1 for no additional flags.

```
int syscall(SYS_perf_event_open, struct perf_event_attr *attr, pid_t pid, int cpu, int
    group_fd, unsigned long flags);
```

This system call creates and returns the Perf file descriptor through which, our tool interacts with the Perf subsystem.

Using the Perf file descriptor acquired before our tool maps two buffers AUX and Data(later refered to as Base) in its memory, required by the `perf_event` counter to provide the performance information. The AUX buffer is where the kernel exposes control flow packets Intel PT captures, whereas the Data buffer contains sideband information such as image changes that are necessary for decoding the trace. It is required that the size of both buffers must be a power of two of the size of the memory page according to `perf_event_open(2)`. The Data buffer requires one additional page to contain the `perf_event_mmap_page`, a metadata page that contains various bits of information such as the beginning of the buffers.

The code snippets below shows the setup process of what we call a collector object. The `perf_ctx` struct, represents the collector, a grouped list of variables required for the Intel PT performance capturing. Such information include the Perf file descriptor obtained previously, pointers that hold the memory address of the actual AUX and Data buffers and variables to store their corresponding size in bytes.

```
//Stores all information about the collector.
struct perf_ctx
{
    int perf_fd;          // File descriptor used to talk to the perf API.
    void *aux_buf;        // Pointer to the start of the the AUX buffer.
    size_t aux_bufsize;   // The size of the AUX buffer's mmap(2).
    void *base_buf;       // Pointer to the start of the base buffer.
    size_t base_bufsize;  // The size the base buffer's mmap(2).
}tr_ctx;
```

First we create a collector object named `tr_ctx`. We then obtain the page size of the current host system using `getpagesize()` call and store that into `page_size` variable. The `getpagesize()` function returns the size of a memory page in bytes. Using that information we calculate the size of the buffer that will be used to store performance event data. It multiplies the value of `tr_conf->data_bufsize`(which is the desired buffer size in pages) by the page size and adds an extra page for the header as explained above. The result is stored in `tr_ctx->base_bufsize`. Then we map the memory for the base buffer using the mmap() system call. Mmap allows a process to map a range of virtual memory addresses to a file or device. It can map memory directly in the kernel, providing a the performance advantage required for Perf Event performance monitoring. The NULL argument indicates that the kernel should choose the address at which to create the mapping. The `tr_ctx->base_bufsize` argument is the length of the mapping, and the `PROT_WRITE` argument specifies that the memory can be written to. The `MAP_SHARED` flag indicates that the mapping should be shared with other processes. The `tr_ctx->perf_fd` argument is a file descriptor that refers to a Perf event, and 0 is the offset within the file descriptor where the mapping should begin. The result of the mmap() call is stored in `tr_ctx->base_buf`. The same procedure is followed for the AUX buffer but with the specific buffer size and `PROT_READ` flag it requires.

```c

int page_size = getpagesize();

tr_ctx->base_bufsize = (1 + tr_conf->data_bufsize) * page_size;
tr_ctx->base_buf = mmap(NULL, tr_ctx->base_bufsize, PROT_WRITE, MAP_SHARED, tr_ctx
    ->perf_fd, 0);

if (tr_ctx->base_buf == MAP_FAILED)
    ...

// Populate the header part of the base buffer.
struct perf_event_mmap_page *base_header = tr_ctx->base_buf;

base_header->aux_offset = base_header->data_offset + base_header->data_size;
base_header->aux_size = tr_ctx->aux_bufsize = tr_conf->aux_bufsize * page_size;

// Allocate the AUX buffer.
tr_ctx->aux_buf = mmap(NULL, base_header->aux_size, PROT_READ | PROT_WRITE,
                       MAP_SHARED, tr_ctx->perf_fd, base_header->aux_offset);

if (tr_ctx->aux_buf == MAP_FAILED)
    ...
```

Ioctl stands for "input-output control" and is a system call that is used to perform device-specific operations that cannot be done through standard file operations. In the context of Perf event tracing, ioctl can be used to enable, reset, and disable tracing. For example, `ioctl(tr_ctx->perf_fd, PERF_EVENT_IOC_ENABLE, 0)` is used to enable tracing by setting the `tr_ctx->perf_fd` file descriptor to the `PERF_EVENT_IOC_ENABLE` operation, which tells the kernel to start tracing. Similarly, `PERF_EVENT_IOC_RESET` can be used to reset the trace, and `PERF_EVENT_IOC_DISABLE` can be used to disable tracing. These ioctl operations allow for fine-grained control over Perf event tracing . The code snippet below is taken from the collection phase loop of our tool and shows how ioctl is used to reset, start and stop Intel Pt tracing in between system calls of our target application.

```c
ioctl(tr_ctx->perf_fd, PERF_EVENT_IOC_RESET, 0); //Resets the event count.
ioctl(tr_ctx->perf_fd, PERF_EVENT_IOC_ENABLE, 0); //Start Intel PT capturing

/*
* Signal tracee to execute until next system call entry
*/

ioctl(tr_ctx->perf_fd, PERF_EVENT_IOC_DISABLE, 0); //Stop Intel PT capturing
```

4.3 Decode Intel Processor Trace

The Intel PT Decoder Library or in short, Libipt[**?**], is Intel's reference implementation for decoding Intel PT[6]. It can be used as a standalone library or it can be partially or fully integrated into a tool. The Libipt decoder library provides multiple layers of abstraction ranging from packet encoding and decoding to full execution flow reconstruction.

Our tool, performs control flow analysis on the executed Tracee's instructions. As a result, it is necessary to decode the control flow packets that AUX buffer contains. Libipt library provides an instruction decoder which we allocate by configuring the `pt_config` object. The `pt_config` structure defines an Intel Processor Trace encoder or decoder. In order to configure `pt_config` for decoding one has to provide information such as the size, beginning and ending addresses of the buffer containing the Intel Pt trace captured, in our case the AUX trace buffer, as well as the CPU identifier which indicates Libipt the processor on which the trace has been collected. In order to dynamically obtain the CPU information Libipt provides the `pt_cpu_read` function which takes the address of the CPU variable in the `pt_config` object and sets it to the correct value. The `pt_cpu_errata()` function enables workarounds for known errata for the processor defined by its family/model/stepping in its CPU argument.(refer to `intel-pt.h`). The code snippet below shows everything mentioned above.

```
struct pt_config config;
memset(&config, 0, sizeof(config));
config.size = sizeof(config);
config.begin = tr_ctx->aux_buf;
config.end = tr_ctx->aux_buf + tr_ctx->aux_bufsize;

int rv = pt_cpu_read(&config.cpu);
if (rv != pte_ok)
    ...

// Work around CPU bugs.
if (config.cpu.vendor){
    rv = pt_cpu_errata(&config.errata, &config.cpu);
    ...
}
```

The pt_config configuration object is required for the allocation of a decoder. The decoder object contains all necessary information libipt needs to decode our trace, including but not limited to, the execution mode(x86/x86-64), address space, current decoding instruction and the image of the Tracee. After instantiating the decoder needs to be synchronized. Synchronization is necessary for the decoder to find the first event of the Intel PT trace, Perf Events written in the AUX buffer. Synchronization is achieved by calling pt_insn_sync_forward() along with the pointer to the instruction decoder object.

```
struct pt_insn_decoder *decoder = NULL;

// Instantiate a decoder.
decoder = pt_insn_alloc_decoder(&config);
if (decoder == NULL)
    ...

// Sync the decoder.
*decoder_status = pt_insn_sync_forward(decoder);
if (*decoder_status == -pte_eos)
    ...
```

In addition to Intel PT configuration, the instruction flow decoder, needs to know the memory image for which Intel PT has been recorded. This is necessary for control flow reconstruction, as Libipt needs to associate the trace with the corresponding instructions found in the ELF file, as well as, for error checking associated with decoding failures, due to an instruction pointer lying outside of the traced memory image. The image is a collection of contiguous, non-overlapping memory regions, called sections that the decoder stores in a pt_image object, Libipt provides. In order to populate the image object, the Tracee's ELF file is loaded from the disk and repeated calls to pt_image_add_cached(), are made, one for each section the ELF file contains, in order add it to the image. After the image is populated the pt_insn_set_image function is called to associate the image with the decoder object. The code for the previous image population process is provided for us by Libipt sample tools[4] in the load_elf.c file. In order to populate the image object we simply call the load_elf() function with the pt_image_section_cache, the pt_image objects as well as the absolute path to the target application and its base address which we extract using extract_base() also provided. For the decoding to be successful, the target application should be statically linked, so that, all the code, for all routines called by it are self-contained. Failure to do so, will result in "No Map" errors, since the decoder will be looking to associate with code that simply don't exist, in the pt_image object. The following code snippet shows the implementation of everything mentioned before.

```
// Build and load a memory image from which to recover control flow.
struct pt_image *image = pt_image_alloc(NULL);
if (image == NULL)
    ...

// Use image cache to speed up decoding.
struct pt_image_section_cache *iscache = pt_iscache_alloc(NULL);
if (iscache == NULL)
    ...

int64_t base;
base = 0ull;      /*The first (lowest) LOAD segment's virtual address is the
    default load base of the file
                  */
int errcode = extract_base(current_exe, &base);
if (errcode < 0)
    ...

errcode = load_elf(iscache, image, current_exe, base, "ptxed_util");
rv = pt_insn_set_image(decoder, image);
if (rv < 0)
    ...
```

Libipt comes with a set of sample tools built on top of it, that serve as a starting point for the integration of the library in our tools. Functionalities such as loading an ELF file or decoding an Intel Pt trace to assembly code, are integrations of such sample codes in our tool. Thus the reader is advised to seek help in the library's repository for further documentation of such functionalities.

The decoder is now initialised and contains all the information required to decode the captured Intel Pt trace. Instructions can now be decoded in execution flow order. The following code snippet shows a stripped down example of the decoding loop which decodes the instructions in the trace.

Firstly, `drain_events_insn()` is called with the decoder and status (an integer variable that the functions modifies depending on the events) parameters to retrieve events that have been stored in the decoder's queue in order to be handled appropriately. These can range from tracing start and stop events, overflow events, and certain exceptions. If there is an error during this process, the decoding loop is terminated. Next, we check whether the end-of-stream (EOS) flag has been set(signifying the reach of the end of the stream) using the `pts_eos` flag in the status variable. If the EOS flag is set the loop is terminated. The loop then calls `pt_insn_next()` with parameters the decoder, insn object(the `pt_insn` object created before, used to store decrypted instructions) and `sizeof(insn)` which specifies the size of the insn object in bytes. This function fetches the next instruction from the trace and decode it into the insn object. Finally it returns the status of the instruction decode operation, which can be a positive number indicating success, or a negative number indicating an error. The decoded instructions are then stored in the execInst array for the control flow analysis.

```
struct pt_insn insn;
uint64_t offset;

//Decoder loop stripped down example
for (;;)
{
    status = drain_events_insn(decoder, status);
    if (status < 0)
        ...

    if (status & pts_eos)
        break;

    //Fetch next instruction from trace
    status = pt_insn_next(decoder, &insn, sizeof(insn));
    if (status < 0)
        ...

    execInst[counter] = insn;
    counter++;
}
```

4.4 Control flow analysis

When a program is undergoing a ROP attack, the attacker manipulates the program's execution flow through the use of gadgets. These gadgets are small code snippets that are already present in the program's memory and are chained together by the attacker. Before ROP code starts executing, the register that holds the stack pointer is set to the beginning of the ROP payload. Each gadget ends with a return instruction, which advances the stack pointer to the address of the next gadget and transfers control to it.

Our detection is based on heuristics regarding the number of call and return instructions executed in the depth of analysis. In order to infer whether the Tracee is under attack or not, we observed that one is required to examine, in most cases, only the control transfers that occur within the final stages of the execution path leading to the system call. System calls take parameters to perform their task. Those parameters are passed by writing them in the appropriate registers before making the actual call. As a result the attacker needs to craft a chain of gadgets to be executed, that deterministically set the values of the registers to those of the parameters to be past. The default analysis depth is set to 100 preceding instructions leading to the system call, almost 3x times more compared to other approaches, this can be configured to any amount but from our observations ROP payload complexity increases drastically the further the gadgets are executed from the system call, since the values of the registers can change. Thus a default analysis depth of a 100 instructions is enough for most cases and even more complex ROP attacks.

Having stored the execution flow that led to the system call the Tracee is about to take. Our tool analyses that trace starting from the syscall and moving backwards, keeping track of the number of executed call and return instructions. Our experiments showed that benign system calls have a light imbalance between the number of call and return instructions, whilst malicious execution tends to have a much greater number of return instructions compared to calls. From our testings we noticed that benign system calls don't tend to have a Call-Ret imbalance greater than 10.

The `pt_insn` object contains an `enum pt_insn_class iclass` variable that indicates the instruction class each object contains. Libipt sets that variable for us during `pt_insn_next()` call. Using that field, our tool keeps track of the number of executed call and return instructions. By the end of the execution flow analysis, if the number of calls are greatly imbalanced(more than 10) to the number of return instructions, the analysed trace is considered malicious, the Tracee process is killed and the user is notified.

The Figure 4.2 below shows an example of a trace leading to a benign system call on the left and a trace of a malicious system call on the right. To print the decoded instructions in assembly, our tool makes use of the Intel X86 Encoder Decoder(XED)[7] library along with a partial integration of the `ptxed` Libipt sample tool. As we can see on the bottom of Figure 4.2 after analysing the benign system call our tool found a Call-Ret imbalance of 1, meaning there was only 1 return instruction for which it could not find a call which the tool interprets as normal behaviour. Whilst on the other hand, the analysis of the flow of execution leading to the malicious system call led to the discovery of 51 unmatched return instructions, which are way more than the limit set, thus triggering the detection mechanism.

```
00000000004130ec call 0x4143d0
00000000004143d0 nop edx, edi
00000000004143d4 mov rax, qword ptr [rdi+0x18]
00000000004143d8 mov qword ptr fs:[0x2f8], rax
00000000004143e1 mov eax, dword ptr [rdi+0x10]
00000000004143e4 mov byte ptr fs:[0x972], al
00000000004143ec cmp eax, 0x1
00000000004143ef jz 0x4143f8
00000000004143f1 ret
00000000004130f1 mov rax, qword ptr [rsp+0x38]
00000000004130f6 sub rax, qword ptr fs:[0x28]
00000000004130ff jnz 0x413195
0000000000413105 add rsp, 0x48
0000000000413109 mov eax, r12d
000000000041310c pop rbx
000000000041310d pop rbp
000000000041310e pop r12
0000000000413110 pop r13
0000000000413112 pop r14
0000000000413114 pop r15
0000000000413116 ret
000000000040a9ea add rbx, 0x8
000000000040a9ee cmp rbx, r12
000000000040a9f1 jb 0x40a9e8
000000000040a9f3 mov edi, ebp
000000000040a9f5 call 0x4466e0
00000000004466e0 nop edx, edi
00000000004466e4 mov r8, 0xffffffffffffffb8
00000000004466eb mov esi, 0xe7
00000000004466f0 mov edx, 0x3c
00000000004466f5 jmp 0x44670d
000000000044670d mov eax, esi
000000000044670f syscall
```

Benign System call
Imbalance: 1 unmatched return instruction

```
000000000044fc47 pop rax
000000000044fc48 ret
00000000004523b5 mov qword ptr [rsi], rax
00000000004523b8 ret
0000000000409f1e pop rsi
0000000000409f1f ret
000000000043e999 xor rax, rax
000000000043e99c ret
00000000004523b5 mov qword ptr [rsi], rax
00000000004523b8 ret
0000000000401eef pop rdi
0000000000401ef0 ret
0000000000409f1e pop rsi
0000000000409f1f ret
0000000000485a6b pop rdx
0000000000485a6c pop rbx
0000000000485a6d ret
000000000043e999 xor rax, rax
000000000043e99c ret
0000000000478130 add rax, 0x1
0000000000478134 ret
0000000000478130 add rax, 0x1
.
.
.
0000000000478130 add rax, 0x1
0000000000478134 ret
0000000000478130 add rax, 0x1
0000000000478134 ret
0000000000401ca4 syscall
```

Malicious system call
Imbalance: 51 unmatched return instructions

Figure 4.2: Execution flow example between a benign and a malicious system call for a control flow analysis configured to analyse 100 preceding instructions leading to the system call.

4.5 Additional functionalities

Our tool comes with a help menu which allow users to quickly access information on how to use the tool and what options and arguments are available. The Listing 4.1 below shows the help menu the user is expected to see when they run the program with -h/–help argument.

The `--depth` option followed by a numeric value the user provides changes the default analysis limit from 100 to the provided value.

The `--pinfo` option prints the Perf Events file descriptor acquired and the configured buffer sizes of both AUX and Base buffers.

The `--pinst` option prints the traced instructions, one instruction per line, in x86[-64] assembly language. This is often used for visual inspection of the traced flow of execution. Is often paired with the `--step` option described later.

The `--pbuff` option writes the contents of AUX and BASE buffers to the disk. This is useful for further analysis outside of our tool.

The `--praw` option writes the decoded instructions in raw hex format to buffer.out file on the disk. This is useful for further analysis outside of our tool.

The `--psyscall` option prints the system call chain of the target application along with the passed arguments for each call.

The `--step` option is used to pause the execution of the program before a system call is executed and wait for the user to press a key to proceed to the next step. This is often paired with `--pinst` option and allows the user to see visually the executed instructions before a system call is executed.

The `--panalysetime` prints the total time the tool takes to record, decode and analyse the target application.

```
usage: ./a.out [<options>] [<Path to Tracee elf file> + arguments]

options:

--depth [Number of Instructions]        preceding number of instructions to check
--pinfo                                 print Intel Pt information
--pinst                                 print traced instructions in x86[-64]
--pbuff                                 print AUX and Base buffers to file
--praw                                  print raw instructions to buffer.out file
--psyscall                              print system call chain
--step                                  step through the syscalls
--panalysetime                          print analysis time
```

Listing 4.1: Help menu

Disassembling of the trace is performed with a stripped down integration of "ptxed" sample tool provided with libipt library. Ptxed uses Intel's XED (X86 Encoder Decoder) library used for encoding and decoding X86 (IA32 and Intel64) instructions, to provide a disassembly of the trace. The XED decoder takes sequences of 1-15 bytes along with machine mode information found in the `pt_insn` object and produces a data structure describing the opcode, operands, and flags of the decoded instruction.

System call chain reconstruction is possible using ptrace. Upon system call entry the tool gathers the content of the registers and prints a representation of it, before its executed.

Chapter 5

Evaluation

In this section we present the results of the experimental evaluation of our tool in terms of runtime overhead and effectiveness on real world applications.

Experiments were performed on the following machine

	Description
CPU	Intel(R) Core(TM) i7-7700 CPU @ 3.60GHz
Memory	16GiB DDR4 2400 MHz
Disk	WDC WD5000AZLX-6 HDD
OS	Ubuntu 22.04.1 LTS

Table 5.1: System specifications

5.1 Runtime Overhead

The runtime overhead tests evaluate the CPU performance impact of eavesdROP on the tracee application. To evaluate the computation performance of the analysis, eavesdROP integrates a timing mechanism that keeps track of the time taken from creation to termination(or detection) of the target application. To evaluate the time taken for the target application to run in the native environment, the `time` linux tool was used. Time tool returns the total time a command takes to run in seconds. The results are illustrated in Table 5.2.

In order to evaluate the runtime performance of our tool, we decided to conduct tests on the 'coreutils-8.32' package of basic Unix utilities. We made this choice because the source code for this package is readily available which allowed us to compile the applications based on our needs(ex. statically). Additionally the low number of system calls each application makes affected our choice since, during testing, we noticed that our tools does not handle efficiently large applications with many system calls.

The table below shows the name and version of the application used for testing, its size in megabytes, the benchmark, meaning the task for which the application was evaluated, the number of system calls that our tool analysed for each test. The native run time, meaning the real time it takes for the application to execute on the Linux system without any additional modifications or optimizations, measured in seconds using 'time' tool. The analysis time which indicates the total time in seconds our tool took to dynamically analyse the application. Finally the performance column shows the ratio of the tool's analysis time compared to the native run time. Each test was performed 5 times, and an average value is presented to eliminate any additional external overheads that may have influenced the results.

We chose to evaluate the runtime overhead this way in order to allow for easy comparison between the other ROP defending approaches mentioned in 'Related Work' chapter. Additionally the reason we mentioned the benchmark and number of system calls the target application makes, is because the two are highly related since the type of benchmark directly affects the number of system calls, which with their turn also affect the analysis time.

Program	**Size(MB)**	**Benchmark**	**Number of Syscalls**	**Native Run**	**Analysis**	**Performance**
sha256sum 8.32	5,0MB	Compute&Check(5,0MB file)	190	0.013s	4.205s	323x
ls 8.32	1,5MB	list directory	32	0.005s	2.348s	470x
cp 8.32	1,3MB	copy(1,5MB file)	62	0.007s	4.719s	674x
gzip 1.10	1,3MB	zip(1,5MB file)	112	0.004s	4.403	1100x

Table 5.2: Runtime Overhead Performance

The above results indicate that eavesdROP analysis introduces a huge performance loss. More specifically, our approach is on average 641 times slower than the native run. This performance loss, renders eavesdROP unusable for real time analysis of applications. We justify this huge performance losses on the approach we followed to utilise our original idea. More specifically our choice to use instruction flow decoding instead of block decoding when using Libipt and the use of ptrace in order to control the target application's execution. These issues are later discussed in the 'Limitations' and 'Conclusion' chapters along with possible solutions to improve and make eavesdROP ideal for real time analysis.

5.2 Effectiveness

The effectiveness tests aim to determine whether our tool can effectively detect and if so, protect applications from Ret-type ROP attacks. Due to the poor run time performance we were only limited to applications that use a small number of system calls. For this reason we were only able to test eavesdROP on the applications mentioned above and a custom made vulnerable application we refer to as ’binTest’ which copies data read from a file into a much smaller allocated buffer.

The table below shows the applications for which eavesdROP was tested. The ’exploit type’ column indicates whether the target application was attacked(method of attack) or not. The analysis depth shows the number of preceding instructions leading to each system call for which the target application was analysed, in order to determine if it was undergoing an attack. In our case the value of the depth was kept to the default value 100 as indicated. Lastly the ’Detected’ column shows whether our tool detected and protected the target application from an attacks (indicated by a check mark) or not(indicated by an X if the tested application was attacked and the tool failed to detect it, or a pass if the application was not undergoing an attack and our tool positively recognised that).

Program	**Exploit Type**	**Analysis Depth**	**Detected**
binTest	ROP	100	✓
sha256sum 8.32	NO	100	pass
ls 8.32	NO	100	pass
cp 8.32	NO	100	pass
gzip 1.10	NO	100	pass

Table 5.3: Effectiveness Accuracy

As we can see from the table above eavesdROP successfully detected all tested applications. Ideally in this part we would test eavesdROP on known vulnerabilities of real life applications with the corresponding ROP attacks but due to its run time overhead, that is not feasible.

Chapter 6

Related Work

6.1 Other ROP defending approaches

The table below compares eavesdROP with other tools that implement different approaches in order to detect ROP attacks[12, 13, 14, 17, 19, 24, 27]. The first column indicates the name of the approach, the 'ROP Type' column shows what kind of ROP attacks the listed tool can detect. Return oriented programming[3, 20] is often used as a general name to describe all attacks that use existing code to exploit a vulnerability, other such attacks are Jump-Oriented Programming(JOP)[26] and Call-Oriented Programming(COP)[21]. For this reason the type is listed to show the capabilities of each tool. The 'No Source Code' columns indicates whether the tools requires the source code of the target application in order to perform the analysis. In a similar way the 'No Binary Rewriting' column indicates whether the tool requires to make modifications to the binary of the application, this is often seen on approaches that perform some kind of binary instrumentation to place hooks. 'Run-time Efficiency' shows if the tool affects the native run time of the target application when under analysis so that it renders it unusable(the research papers of the other approaches set the barrier to 20% overhead). Finally, 'Not branch limited' column indicates whether the depth of the analysis is limited. Kbouncer[19], ROPGuard[12] and ROPecker[27] rely on the Last Branch Recording (LBR) feature of Intel and AMD processors which limits the depth of their analysis since LBR feature is stack limited to only 16(32 for newer CPUs) entries. LBR-based solutions are vulnerable to history-flushing attacks[2, 22], where the payload on purpose includes dummy branch instructions to flush LBR entries in order to avoid detection.

	ROP Type	No Source Code	No Binary Rewriting	Run-time Efficiency	Not Branch Limited
DROP[17]	Ret-based	✓	X	X	✓
ROPDefender[14]	Ret-based	✓	X	X	✓
ROPGuard[12]	Ret-based	✓	X	✓	X
Return-less Kernel[13]	Ret-based	X	✓	✓	✓
CFLocking[24]	All	X	✓	✓	✓
Kbouncer[19]	All	✓	X	✓	X
ROPecker[27]	All	✓	✓	✓	X
eavesdROP	Ret-based	X	✓	X	✓

Table 6.1: Comparison Between Other ROP Approaches

As the above table indicates, eavesdROP is not better than the other approaches. EavesdROP whilst requires no binary instrumentation and is not branch limited, it lacks the ability to perform its analysis without requiring the targets application source code. This is due to the requirement of statically linked binaries. Moreover as shown on the previous chapter our tool is far from run time efficient. All these drawbacks are further discussed in the 'Limitations' and 'Conclusion' chapters where possible solutions are suggested.

A. Address Randomization

Address Space Layout Randomization (ASLR) is a security technique proposed to defend against Return-Oriented Programming (ROP) attacks. ASLR works by randomly arranging the locations of key components of a program's address space, such as the stack, heap, and code sections, each time the program is executed. This makes it difficult for an attacker to predict the location of specific gadgets or other components in the address space, and therefore makes it more difficult for the attacker to construct a successful ROP attack.

However, attackers have developed techniques to bypass ASLR[23]. One such technique is memory disclosure, where the attacker leverages a vulnerability in the program to leak information about the memory layout of the program at runtime. Another technique is brute force, where the attacker attempts to repeatedly execute the program with different memory layouts until they find one that works for their ROP attack. Additionally, attackers have developed methods to defeat specific types of ASLR, such as kernel ASLR, which randomizes the location of the kernel in memory, by identifying and exploiting weaknesses in the implementation of these techniques.

B. Control Flow checks

Control Flow Integrity (CFI)[16] is a security technique designed to defend against code reuse attacks such as Return-Oriented Programming (ROP) attacks. CFI works by enforcing constraints on the control flow of a program, such that the program can only execute instructions in a valid order according to its control flow graph. CFLocking[24] limits the number of abnormal control flow transfers by recompiling the source code of the target application. The Return-less Kernel[13] approach implements methods through a compiler and is designed to eliminate the use of the ret opcode in the kernel image. Instead of using the stack to store control data, this approach stores the control data in a separate buffer.

Control Flow Integrity (CFI) is accomplished by adding metadata to the binary code of the program, such as information about the valid targets of a particular function call. At runtime, the CFI system checks the metadata to ensure that the program is following a valid control flow path or not.

However, there are several drawbacks to CFI as a defense mechanism. One of the primary drawbacks is the performance overhead of CFI, as it requires additional runtime checks and metadata processing. Additionally, CFI can be bypassed[9] by attackers who are able to control the flow of the program in ways that are still considered valid by the CFI system. For example, some ROP attacks use unaligned gadgets, which are sequences of instructions that do not start at the beginning of an instruction boundary, and therefore do not match the expected control flow path. These types of attacks can be difficult for CFI systems to detect, as they may be interpreted as legitimate control flow changes by the system.

C. Binary instrumentation

Instrumentation-based approaches are a class of security techniques designed to defend against code reuse attacks such as Return-Oriented Programming (ROP). These techniques work by inserting additional code(using tools such us PIN[15]), known as "check code," into the binary code of a program that checks for violations of the expected control flow. ROPDefender[14] and DROP[17] use binary instrumentation to help with the detection of ROP attacks.

At runtime, the check code monitors the execution of the program and verifies that the program is following a valid control flow path. If the check code detects a deviation from the expected control flow, it can terminate the program or take other protective measures to prevent further damage.

One drawback of instrumentation-based approaches is the overhead they impose on program execution. Because the check code must execute alongside the program code, it can slow down the performance of the program and increase its memory footprint. Additionally, attackers can attempt to evade the check code by manipulating the program in ways that do not trigger the expected checks. For example, attackers may modify the code in such a way that the check code is not executed, or they may attempt to bypass the checks by exploiting weaknesses in the instrumentation mechanism itself.

Chapter 7

Limitations

7.1 eavesdROP tool limitations

The Intel Processor Trace feature of Intel processors allows for a transparent, non depth limited tracing. Whilst Intel PT is designed to have a low overhead on system performance and provides a fast and efficient way to trace program execution without significant performance impact(in the range of 1-5%), our implementation lacks the run time efficiency other ROP defending approaches provide. This poor performance is mainly the result of the decoding and execution control process. The trace contains several packets such as the PSB(Packet Stream Boundary) a synchronization packet that provides a starting point for decoding the trace. The packets within a packet stream must be decoded serially and in the correct order. This is because the packets are dependent on each other and may reference information from previous packets in the stream. If the packets are decoded out of order or with missing packets, the decoded information may be incorrect or incomplete. As a result, one is bound to decode the whole trace only to analyse a very small part leading to the system call. Libipt decoder library provides several layers of abstraction for decoding the trace, our implementation uses instruction flow. Instruction flow layer deals with the execution flow on the instruction level and provides a simple API for iterating over instructions in execution order. This layer of decoding is generally slow compared to the other options Libipt provides, such as the block layer, a much faster approach that requires a small amount of post-processing. Additionally our approach to use ptrace facility to control the execution of the target application introduces the largest amount of run time overhead.

The current implementation is limited in its ability to analyze dynamically linked binaries, as it was not specifically designed to load and decode dynamically linked sections. As a consequence, this approach can only effectively analyze statically compiled executables.

Chapter 8

Conclusion

We showcased eavesdROP, a Ret-type Return Oriented Programming(ROP), non-branch limited, attack detection and prevention tool, build on top of Intel Processor Trace feature of recent processors, that provides control flow tracing of a process. EavesdROP, designed for Linux machines, uses the ptrace facility along with Intel PT feature and Libipt, Intel's reference implementation library for decoding Intel PT traces, in order to perform dynamic, transparent, variable depth, Call-Ret imbalance heuristic analysis of a statically compiled target application in pursuance to determine whether its undergoing a ROP attack. EavesdROP requires no source code and does not perform any modifications to the protected application whatsoever. Finally we showed that our prototype implementation is able to effectively protect against ROP exploits with the only drawback the run time overhead it introduces.

As of future work, the tool could be optimised to reduce the significant runtime performance it introduces and possibly make it run time efficient. Optimisations can be made to the decoding process by choosing block layer decoding approach which is much faster than the existing approach we are using. Furthermore faster execution control methods can be put into practise to stop the target application upon system call entry, required for analysis. A good mechanism would be some kind of binary instrumentation that places hooks which intercept the normal flow of execution and allow our tool to perform its analysis. Finally, functionality can be added to support tracing of dynamically linked binaries and more extensive evaluations could be carried on real applications, to ensure that is able to detect more complex ROP payloads successfully.

Bibliography

[1] Starr Andersen and Vincent Abella. Data execution prevention. changes to functionality in microsoft windows xp service pack 2, part 3: Memory protection technologies, 2004.

[2] Nicholas Carlini and David Wagner. {ROP} is still dangerous: Breaking modern defenses. In *23rd SecuritySymposium(Security 14)*, pages 385–399, 2014.

[3] Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, and Marcel Winandy. Return-oriented programming without returns. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*. ACM, 2010.

[4] Intel Corporation. libipt: How to build. `https://github.com/intel/libipt/blob/master/doc/howto_build.md`, .

[5] Intel Corporation. libipt: How to capture. `https://github.com/intel/libipt/blob/master/doc/howto_capture.md`, .

[6] Intel Corporation. libipt: How to use libipt. `https://github.com/intel/libipt/blob/master/doc/howto_libipt.md`, .

[7] Intel Corporation. xed: The x86 encoder decoder. `https://github.com/intelxed/xed`, .

[8] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. Stackguard: automatic adaptive detection and prevention of buffer-overflow attacks. In *USENIX security symposium*, volume 98, pages 63–78. San Antonio, TX, 1998.

[9] Lucas Davi, Ahmad-Reza Sadeghi, Daniel Lehmann, and Fabian Monrose. Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection. In *23rd $SecuritySymposium($ Security 14)*, pages 401–416, 2014.

[10] Dai Zovi Dino A. Practical return-oriented programming. 2010.

[11] Ulfar Erlingssoni. Low-level software security: Attacks and defenses. 2007.

[12] F. Ivan. Runtime prevention of return-oriented programming attacks. `https://code.google.com/p/ropguard/`.

[13] X. Jiang M. Grace J. Li, Z. Wang and S. Bahram. Defeating Return-oriented Rootkits with "Return-Less" Kernels. In *Proceedings of the 5th European Conference on Computer Systems*, 2010.

[14] A.-R. Sadeghi L. Davi and M. Winandy. ROPdefender: A Detection Tool to Defend Against Return-oriented Programming Attacks. In *Proc. of the 6th ACM Symposium on Information, Computer and Communications Security*, 2011.

[15] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. *Acm sigplan notices*, 40(6):190–200, 2005.

[16] U. Erlingsson M. Abadi, M. Budiu and J. Ligatti. Control-flow integrity. In *Proceedings of the 12th ACM conference on Computer and Communications Security*. ACM, 2005.

[17] X. Shen X. Yin B. Mao P. Chen, H. Xiao and L. Xie. DROP: Detecting Return-Oriented Programming Malicious Code. In *Proc. of the 5th International Conference on Information Systems Security*, 2009.

[18] Vasilis Pappas. kBouncer: Efficient and Transparent ROP Mitigation.

[19] Vasilis Pappas, Michalis Polychronakis, and Angelos D Keromytis. Transparent rop exploit mitigation using indirect branch tracing. In *USENIX Security Symposium*, pages 447–462, 2013.

[20] A. Dmitrienko A.-R. Sadeghi H. Shacham S. Checkoway, L. Davi and M. Winandy. Return oriented programming without returns. In Proceedings of. The Geometry of Innocent Flesh on the Bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security*. ACM Press, 2007.

[21] Niksefat-S. Rostamipour Sadeghi, A. Pure-call oriented programming (pcop): Chaining the gadgets using call instructions. *Journal of Computer Virology and Hacking Techniques*, 2018. doi: 10.1007/s11416-017-0299-1.

[22] Felix Schuster, Thomas Tendyck, Jannik Pewny, Andreas Maaß, Martin Steegmanns, Moritz Contag, and Thorsten Holz. Evaluating the effectiveness of current anti-rop defenses. In *Research in Attacks, Intrusions and Defenses: 17th International Symposium, RAID 2014, Gothenburg, Sweden, September 17-19, 2014. Proceedings 17*, pages 88–108. Springer, 2014.

[23] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM conference on Computer and communications security*, pages 298–307, 2004.

[24] X. Jiang T. Bletsch and V. Freeh. Mitigating Code-reuse Attacks with Control-flow Locking. In *Proceedings of the 27th Annual Computer Security Applications Conference*, 2011.

[25] PaX Team. Pax address space layout randomization (aslr). *http://pax. grsecurity. net/docs/aslr. txt*, 2003.

[26] Vince Freeh Tyler Bletsch, Xuxian Jiang and Zhenkai Liang. Jump-oriented programming: A new class of code-reuse attack. In *Proceedings of the 6th Symposium on Information, Computer and Communications Security (ASIACCS)*, 2011.

[27] Yu MIAO Xuhua DING Robert H. DENG Yueqiang CHENG, Zongwei ZHOU. Ropecker: A generic and practical approach for defending against rop attack. In *NDSS Symposium 2014: Proceedings of the 21st Network and Distributed System Security Symposium, San Diego, February 23-26*, 2014.

Appendix A

Source Code

A.1 main.c

```c
#define _GNU_SOURCE

#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/ioctl.h>
#include <sys/ptrace.h>
#include <sys/wait.h>
#include <sys/user.h>
#include <link.h>

#include <time.h>

#include "perf_pt/collect.c"
#include "perf_pt/decode.c"


//Compile
// gcc -L /usr/local/lib/ main.c  -lipt -lxed

#define FATAL(...)                              \
    do                                          \
    {                                           \
        fprintf(stderr, "strace: " __VA_ARGS__); \
        fputc('\n', stderr);                    \
        exit(EXIT_FAILURE);                     \
    } while (0)

// Data,Aux,Trace buffer sizes
#define PERF_PT_DFLT_DATA_BUFSIZE 64
#define PERF_PT_DFLT_AUX_BUFSIZE 1024
#define PERF_PT_DFLT_INITIAL_TRACE_BUFSIZE 1024 * 1024

#define MAXLIST 100

char* parsedArgs[MAXLIST];

```

```
struct perf_collector_config pptConf = {
    .data_bufsize = PERF_PT_DFLT_DATA_BUFSIZE,
    .aux_bufsize = PERF_PT_DFLT_AUX_BUFSIZE,
    .initial_trace_bufsize = PERF_PT_DFLT_INITIAL_TRACE_BUFSIZE};

void write_memory(void *addr, size_t size, char *filename)
{
   void *readout = malloc(size);
   memcpy(readout, addr, size);
   FILE *fd = fopen(filename, "wb");
   fwrite(readout, 1, size, fd);
   fclose(fd);
   free(readout);
}

void print_help()
{
   printf("usage: ./a.out [<Path to Tracee elf file>] [<options>]\n\n");
   printf("options:\n\n");
   printf("--depth [numOfInstructions]          preceding number of instructions to check\n");
   printf("--pinfo                              print Intel Pt information\n");
   printf("--pinst                              print traced instructions in x86[-64]\n");
   printf("--pbuff                              print AUX and Base buffers\n");
   printf("--praw                               print raw instructions in buffer.out file\n");
   printf("--psyscall                           print system call chain\n");
   printf("--step                               Step through the syscalls\n");
   printf("--ptracetime                         print intel Pt trace time and exit\n");
   printf("--panalysetime                       print analysis time\n\n");
   return;
}

int main(int argc, char **argv)
{
   int pArgs=0;

   clock_t begin;
   clock_t end;
   double time_spent;

   if (argc <= 1)
      FATAL("too few arguments: %d", argc);

   if (argc > 1)
   {
      char *arg;

      int i=0;
      for(i=1;i<argc;i++){
         arg = argv[i];

         if(arg[0]!='-')
            break;

         if (strcmp(arg, "--help") == 0)
```

```
        {
            print_help();
            return 0;
            continue;
        }
        if (strcmp(arg, "-h") == 0)
        {
            print_help();
            return 0;
            continue;
        }
        if (strcmp(arg, "--depth") == 0)
        {
            if (argc <= i) {
            fprintf(stderr,
                "--depth: missing argument.\n");
                return 1;
            }
            stats.limited=true;
            stats.depth = atoi(argv[++i]);
            continue;
        }
        if (strcmp(arg, "--pinfo") == 0)
        {
            stats.pinfo = true;
            continue;
        }
        if (strcmp(arg, "--pinst") == 0)
        {
            stats.pinst = true;
            continue;
        }
        if (strcmp(arg, "--pbuff") == 0)
        {
            stats.pbuff = true;
            continue;
        }
        if (strcmp(arg, "--praw") == 0)
        {
            stats.praw = true;
            continue;
        }
        if (strcmp(arg, "--psyscall") == 0)
        {
            stats.psyscall = true;
            continue;
        }
        if (strcmp(arg, "--step") == 0)
        {
            stats.step = true;
            continue;
        }
        if (strcmp(arg, "--panalysetime") == 0)
        {
            stats.panalysetime = true;
            continue;
        }
```

```
            printf("unknown option: %s\n", arg);
            return 0;
        }

        pArgs=i;
    }

    pid_t traceepid = fork();

    switch (traceepid)
    {
    case -1: /* error */
        FATAL("%s", strerror(errno));
    case 0: /* child */
        ptrace(PTRACE_TRACEME, 0, 0, 0);
        /* Because we're now a tracee, execvp will block until the parent
         * attaches and allows us to continue. */
        execvp(argv[pArgs], (argv+pArgs));
        FATAL("%s", strerror(errno));
    }

    // Wait for tracee to stop
    waitpid(traceepid, 0, 0);

    ptrace(PTRACE_SETOPTIONS, traceepid, 0, PTRACE_O_TRACEEXIT);

    int dec_status;
    struct pt_insn_decoder *decoder;
    bool first = true;

    //
    struct perf_ctx *tracer = perf_init_collector(&pptConf, traceepid, &stats);
    if (tracer == NULL)
        printf("Collector error");

    if (stats.pinfo)
    {
        printf("perf_fd %d\n", tracer->perf_fd);
    }

    if (stats.pinfo)
    {
        printf("Aux Buffer size: %ld\n", tracer->aux_bufsize);
        printf("Base Buffer size: %ld\n", tracer->base_bufsize);
    }

    if(stats.panalysetime){
        begin=clock();
    }

    //Main tracing loop
    for (;;)
    {
        ioctl(tracer->perf_fd, PERF_EVENT_IOC_RESET, 0);
        ioctl(tracer->perf_fd, PERF_EVENT_IOC_ENABLE, 0);

```

```
        /* Enter next system call */
        if (ptrace(PTRACE_SYSCALL, traceepid, 0, 0) == -1)
        {
            // Tracee is dead, this is triggered when tracee finish executing
            if (errno == ESRCH)
                break;
            FATAL("%s", strerror(errno));
        }

        if (waitpid(traceepid, 0, 0) == -1)
        {
            // Tracee is dead, this is triggered when tracee finish executing
            if (errno == ESRCH)
                break;
            FATAL("%s", strerror(errno));
        }

        ioctl(tracer->perf_fd, PERF_EVENT_IOC_DISABLE, 0);

        if (stats.psyscall)
        {
            /* Gather system call arguments */
            struct user_regs_struct regs;
            if (ptrace(PTRACE_GETREGS, traceepid, 0, &regs) == -1)
            {
                // Tracee is dead, this is triggered when tracee finish executing
                if (errno == ESRCH)
                    break;
                FATAL("%s", strerror(errno));
            }

            long syscall = regs.orig_rax;
            /* Print a representation of the system call */
            fprintf(stderr, "%ld(%ld, %ld, %ld, %ld, %ld, %ld)\n",
                    syscall,
                    (long)regs.rdi, (long)regs.rsi, (long)regs.rdx,
                    (long)regs.r10, (long)regs.r8, (long)regs.r9);
            if(stats.step){
                 printf("Press any character to continue\n");
                 getchar();
            }
        }

        if (stats.pbuff)
        {
            write_memory(tracer->aux_buf, tracer->aux_bufsize, "aux");
            write_memory(tracer->base_buf, tracer->base_bufsize, "base");
        }

        if (first)
        {
            first = false;
            decoder = init_inst_decoder(tracer->aux_buf, tracer->aux_bufsize, &dec_status,
                  argv[pArgs], &stats);
            if (decoder == NULL)
                printf("error: decoder initialization\n");
        }
```

```
        else
        {
            int dec_status = pt_insn_sync_set(decoder, 0);
            if (dec_status == -pte_eos)
            {
                // There were no blocks in the stream. The user will find out on next
                // call to hwt_ipt_next_block().
                printf("no blocks\n");
            }
            else if (dec_status < 0)
            {
                printf("sync error\n");
            }
        }

        if (!decode_trace(decoder, &dec_status, &stats))
            {
                ptrace(PTRACE_KILL, traceepid, 0, 0);
                return 0;
            }
        if(stats.step){
            printf("Press any character to continue\n");
            getchar();
        }


        /* Run system call and stop on exit */
        if (ptrace(PTRACE_SYSCALL, traceepid, 0, 0) == -1)
        {
            // Tracee is dead, this is triggered when tracee finish executing
            if (errno == ESRCH)
                break;
            FATAL("%s", strerror(errno));
        }
        if (waitpid(traceepid, 0, 0) == -1)
        {
            // Tracee is dead, this is triggered when tracee finish executing
            if (errno == ESRCH)
                break;
            FATAL("%s", strerror(errno));
        }

    } // End loop


    if(stats.panalysetime){
        end=clock();
        time_spent = (double)(end-begin) / CLOCKS_PER_SEC;
        printf("%f second\n",time_spent);

    }

    printf("No attacks found!\n");

    free_insn_decoder(decoder);
    if (!perf_free_collector(tracer))
        printf("error: Freeing Tracer\n");}
```

A.2 collect.c

```c
#define _GNU_SOURCE

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <syscall.h>
#include <sys/mman.h>
#include <inttypes.h>
#include <errno.h>
#include <stdbool.h>
#include <time.h>
#include <intel-pt.h>
#include <linux/perf_event.h>

#define SYSFS_PT_TYPE "/sys/bus/event_source/devices/intel_pt/type"
#define MAX_PT_TYPE_STR 8

#define MAX_OPEN_PERF_TRIES 50000
#define OPEN_PERF_WAIT_NSECS 10000000 // 1/100 of a second.

#define AUX_BUF_WAKE_RATIO 0.5

#ifndef INFTIM
#define INFTIM -1
#endif

/*
 * Stores all information about the collector.
 */
struct perf_ctx
{
    int perf_fd;          // FD used to talk to the perf API.
    void *aux_buf;        // Ptr to the start of the the AUX buffer.
    size_t aux_bufsize;   // The size of the AUX buffer's mmap(2).
    void *base_buf;       // Ptr to the start of the base buffer.
    size_t base_bufsize;  // The size the base buffer's mmap(2).
};

struct stats_config
{
    bool pinfo;
    bool pinst;
    bool pbuff;
    bool praw;
    bool psyscall;
    bool step;
    bool limited;
    bool panalysetime;
    int depth;
} stats;

struct perf_collector_config
{
    size_t data_bufsize;          // Data buf size (in pages).
```

```c
    size_t aux_bufsize;            // AUX buf size (in pages).
    size_t initial_trace_bufsize; // Initial capacity (in bytes) of a
                                   // trace storage buffer.
};

// Private prototypes.
static int open_perf(size_t, pid_t traceepid, struct stats_config *);

// Exposed Prototypes.
struct perf_ctx *perf_init_collector(struct perf_collector_config *, pid_t traceepid,
    struct stats_config *);
bool perf_free_collector(struct perf_ctx *tr_ctx);

/*
 * Opens the perf file descriptor and returns it.
 *
 * Returns a file descriptor, or -1 on error.
 */
static int
open_perf(size_t aux_bufsize, pid_t traceepid, struct stats_config *stats)
{
    struct perf_event_attr attr;
    memset(&attr, 0, sizeof(attr));
    attr.size = sizeof(attr);
    // attr.size = sizeof(struct perf_event_attr);

    int ret = -1;

    // Get the perf "type" for Intel PT.
    FILE *pt_type_file = fopen(SYSFS_PT_TYPE, "r");
    if (pt_type_file == NULL)
    {
        printf("Error: openning perf 'type' file descriptor");
        ret = -1;
        goto clean;
    }
    char pt_type_str[MAX_PT_TYPE_STR];
    if (fgets(pt_type_str, sizeof(pt_type_str), pt_type_file) == NULL)
    {
        printf("Error: reading perf 'type'");
        ret = -1;
        goto clean;
    }
    attr.type = atoi(pt_type_str);
    if (stats->pinfo)
        printf("Intel PT type: %d\n", attr.type);

    attr.config = 0x300e601;

    // Exclude the kernel.
    attr.exclude_kernel = 1;

    // Exclude the hyper-visor.
    attr.exclude_hv = 1;

    // Start disabled.
    attr.disabled = 1;
```

```
    // No skid.
    attr.precise_ip = 3;

    // Notify for every sample.
    attr.watermark = 1;
    attr.wakeup_watermark = 1;

    // Generate a PERF_RECORD_AUX sample when the AUX buffer is almost full.
    attr.aux_watermark = (size_t)((double)aux_bufsize * getpagesize()) *
        AUX_BUF_WAKE_RATIO;

    // Acquire file descriptor through which to talk to Intel PT. This syscall
    // could return EBUSY, meaning another process or thread has locked the
    // Perf device.
    struct timespec wait_time = {0, OPEN_PERF_WAIT_NSECS};

    // pid_t target_tid = syscall(__NR_gettid);
    for (int tries = MAX_OPEN_PERF_TRIES; tries > 0; tries--)
    {
        ret = syscall(SYS_perf_event_open, &attr, traceepid, -1, -1, 0);
        if ((ret == -1) && (errno == EBUSY))
        {
            nanosleep(&wait_time, NULL); // Doesn't matter if this is interrupted.
        }
        else
        {
            break;
        }
    }

    if (ret == -1)
    {
        printf("Error openning perf_event");
    }

clean:
    if ((pt_type_file != NULL) && (fclose(pt_type_file) == -1))
    {
        ret = -1;
    }

    return ret;
}


/*
 * Initialise a collector context.
 */
struct perf_ctx *
perf_init_collector(struct perf_collector_config *tr_conf, pid_t traceepid, struct
    stats_config *stats)
{
    struct perf_ctx *tr_ctx = NULL;
    bool failing = false;

    // Allocate and initialise collector context.
```

```
    tr_ctx = malloc(sizeof(*tr_ctx));
    if (tr_ctx == NULL)
    {
        printf("Error: allocating collector");
        failing = true;
        goto clean;
    }

    // Set default values.
    memset(tr_ctx, 0, sizeof(*tr_ctx));
    tr_ctx->perf_fd = -1;

    // Obtain a file descriptor through which to speak to perf.
    tr_ctx->perf_fd = open_perf(tr_conf->aux_bufsize, traceepid, stats);
    if (tr_ctx->perf_fd == -1)
    {
        printf("Error: obtaining a perf event file descriptor");
        failing = true;
        goto clean;
    }

    int page_size = getpagesize();
    // printf("\n%d\n",page_size);
    tr_ctx->base_bufsize = (1 + tr_conf->data_bufsize) * page_size;
    tr_ctx->base_buf = mmap(NULL, tr_ctx->base_bufsize, PROT_WRITE, MAP_SHARED, tr_ctx
        ->perf_fd, 0);

    if (tr_ctx->base_buf == MAP_FAILED)
    {
        printf("Error: mapping base buffer");
        failing = true;
        goto clean;
    }

    // Populate the header part of the base buffer.
    struct perf_event_mmap_page *base_header = tr_ctx->base_buf;
    base_header->aux_offset = base_header->data_offset + base_header->data_size;
    base_header->aux_size = tr_ctx->aux_bufsize =
        tr_conf->aux_bufsize * page_size;

    // Allocate the AUX buffer.
    //
    // Mapped R/W so as to have a saturating ring buffer.
    tr_ctx->aux_buf = mmap(NULL, base_header->aux_size, PROT_READ | PROT_WRITE,
                           MAP_SHARED, tr_ctx->perf_fd, base_header->aux_offset);
    if (tr_ctx->aux_buf == MAP_FAILED)
    {
        printf("Error: mapping aux buffer");
        failing = true;
        goto clean;
    }

clean:
    if (failing && (tr_ctx != NULL))
    {
        perf_free_collector(tr_ctx);
        return NULL;
```

```
    }
    return tr_ctx;
}

/*
 * Clean up and free a perf_ctx and its contents.
 *
 * Returns true on success or false otherwise.
 */
bool perf_free_collector(struct perf_ctx *tr_ctx)
{
    int ret = true;

    if ((tr_ctx->aux_buf) &&
        (munmap(tr_ctx->aux_buf, tr_ctx->aux_bufsize) == -1))
    {
        printf("Error: unmapping aux buffer");
        ret = false;
    }
    if ((tr_ctx->base_buf) &&
        (munmap(tr_ctx->base_buf, tr_ctx->base_bufsize) == -1))
    {
        printf("Error: unmapping base buffer");
        ret = false;
    }
    if (tr_ctx->perf_fd >= 0)
    {
        close(tr_ctx->perf_fd);
        tr_ctx->perf_fd = -1;
    }
    if (tr_ctx != NULL)
    {
        free(tr_ctx);
    }
    return ret;
}
```

A.3 decode.c

```
#define _GNU_SOURCE

#include <stdio.h>
#include <intel-pt.h>
#include <pt_cpu.h>
#include <stdbool.h>
#include <inttypes.h>
#include <stdint.h>
#include <link.h>
#include <errno.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdbool.h>

#include "ptxed_util.c"
#include "analyse_exec_flow.c"
#include "pt_cpu.c"
#include "pt_cpuid.c"
#include "load_elf.c"

// Storage for executed instructions
struct pt_insn execInst[100000];

// Private prototypes
static int extract_base(const char *, uint64_t *);

// Public prototypes.
void *init_inst_decoder(void *buf, uint64_t len,
                        int *decoder_status,
                        const char *current_exe, struct stats_config *);
bool decode_trace(struct pt_insn_decoder *decoder, int *decoder_status, struct stats_config *);
void free_insn_decoder(struct pt_insn_decoder *);

static int extract_base(const char *arg, uint64_t *base)
{
    char *sep, *rest;

    sep = strrchr(arg, ':');
    if (sep)
    {
        uint64_t num;

        if (!sep[1])
            return 0;

        errno = 0;
        num = strtoull(sep + 1, &rest, 0);
        if (errno || *rest)
            return 0;

        *base = num;
        *sep = 0;
        return 1;
```

```
    }

    return 0;
}


void *
init_inst_decoder(void *buf, uint64_t len,
                  int *decoder_status, const char *current_exe, struct stats_config *
                      stats)
{
    bool failing = false;
    if (stats->praw)
        bufferFd = fopen("buffer.out", "w+");

    struct pt_config config;
    memset(&config, 0, sizeof(config));

    // pt_config_init(&config);

    config.size = sizeof(config);
    config.begin = buf;
    config.end = buf + len;

    // Decode for the current CPU.
    struct pt_insn_decoder *decoder = NULL;
    int rv = pt_cpu_read(&config.cpu);
    if (rv != pte_ok)
    {
        printf("Error: reading cpu");
        failing = true;
        goto clean;
    }

    // Work around CPU bugs.
    if (config.cpu.vendor)
    {
        rv = pt_cpu_errata(&config.errata, &config.cpu);
        if (rv < 0)
        {
            printf("Error: working around bugs");
            failing = true;
            goto clean;
        }
    }

    // Instantiate a decoder.
    decoder = pt_insn_alloc_decoder(&config);
    if (decoder == NULL)
    {
        printf("Error: instantiating decoder");
        failing = true;
        goto clean;
    }

    // Sync the decoder.
    *decoder_status = pt_insn_sync_forward(decoder);
```

```c
    if (*decoder_status == -pte_eos)
    {
        // There were no blocks in the stream. The user will find out on next
        // call to hwt_ipt_next_block().
        goto clean;
    }
    else if (*decoder_status < 0)
    {
        printf("Error: synchronising decoder");
        failing = true;
        goto clean;
    }

    // Build and load a memory image from which to recover control flow.
    struct pt_image *image = pt_image_alloc(NULL);
    if (image == NULL)
    {
        printf("Error: allocating image");
        failing = true;
        goto clean;
    }
    // Use image cache to speed up decoding.
    struct pt_image_section_cache *iscache = pt_iscache_alloc(NULL);

    if (iscache == NULL)
    {
        printf("Error: allocating cache");
        failing = true;
        goto clean;
    }

    int64_t base;
    base = 0ull;

    int errcode = extract_base(current_exe, &base);
    if (errcode < 0)
    {
        printf("Error: Extracting base");
        failing = true;
        goto clean;
    }

    errcode = load_elf(iscache, image, current_exe, base, "ptxed_util");

    rv = pt_insn_set_image(decoder, image);
    if (rv < 0)
    {
        printf("Error: setting image to decoder");
        failing = true;
        goto clean;
    }

clean:
    if (failing)
    {
        pt_insn_free_decoder(decoder);
        return NULL;
```

```
    }
    return decoder;
}

/*
 *
 * Decodes intel PT
 *
 */
bool decode_trace(struct pt_insn_decoder *decoder, int *decoder_status, struct
    stats_config *stats)
{
    xed_state_t xed;
    if (stats->pinst)
    {
        xed_state_zero(&xed);
        xed_tables_init();
    }

    uint64_t offset, sync;

    offset = 0ull;
    int errcode;

    int status = *decoder_status;
    struct pt_insn insn;

    // Used to keep track of the number of instructions
    int counter = 0;

    /* Initialize the IP - we use it for error reporting. */
    insn.ip = 0ull;

    for (;;)
    {
        status = drain_events_insn(decoder, status);
        if (status < 0)
        {
            printf("Drain Events error \n");
            break;
        }

        if (status & pts_eos)
        {
            // printf("[End of trace]\n");
            break;
        }

        errcode = pt_insn_get_offset(decoder, &offset);
        if (errcode < 0)
        {
            printf("Get offset error");
            break;
        }

        status = pt_insn_next(decoder, &insn, sizeof(insn));
        if (status < 0)
```

```
        {
            /* Even in case of errors, we may have succeeded
             * in decoding the current instruction.
             */
            print_insn(&insn, &xed, offset);
            printf("Error␣fetching␣instruction\n");
        }

        execInst[counter] = insn;
        counter++;

        if(counter>99997)
            counter= stats->depth+1;

        if (stats->pinst)
            print_insn(&insn, &xed, offset);

        if (stats->praw)
            print_raw_insn_file(&insn);

    }

    /* We shouldn't break out of the loop without an error. */
    if (!status)
        status = -pte_internal;

    /* We're done when we reach the end of the trace stream. */
    if (status == -pte_eos)
    {
        printf("Error␣with␣end␣of␣trace␣stream\n");
        return false;
    }


    if (!exec_flow_analysis(execInst, counter))
    {
        printf("Rop␣chain␣detected\n");
        return false;
    }
    else
    {
        if (stats->psyscall)
        {
            printf("Syscall␣safe\n");
        }
    }
    return true;
}

/*
 * Free an instruction decoder and its image.
 */
void free_insn_decoder(struct pt_insn_decoder *decoder)
{
    if (decoder != NULL)
    {
        pt_insn_free_decoder(decoder);}}
```

A.4 analyse_exec_flow.c

```c
#define _GNU_SOURCE

#include <stdio.h>
#include <intel-pt.h>
#include <stdbool.h>

bool exec_flow_analysis(struct pt_insn *execInstArr, int instCnt)
{

    int cnt = 1;
    int stop= -1;

    if(stats.limited && stats.depth<instCnt){
        stop = instCnt-stats.depth-1;
    }

    for (int i= instCnt - 1;i >stop; i--)
    {
        switch (execInstArr[i].iclass)
        {
        case ptic_call: // Near (function) call
            cnt--;
            break;
        case ptic_return: // Near (function) return
            cnt++;
            break;
        }
    }
    //printf("Call/Ret Ibalance\n%d\n",cnt);
    if(cnt<10)
        return true;

    return false;
}
```

A.5 ptxed_util.c

```c
#define _GNU_SOURCE

#include <intel-pt.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <inttypes.h>
#include <errno.h>
#include <pt_cpu.h>
#include <xed/xed-interface.h>

FILE *bufferFd;

/* A collection of statistics. */
struct ptxed_stats
{
  /* The number of instructions. */
  uint64_t insn;

  /* The number of blocks.
   *
   * This only applies to the block decoder.
   */
  uint64_t blocks;

  /* A collection of flags saying which statistics to collect/print. */
  uint32_t flags;
};

/*
Private Prototypes
*/
static const char *print_exec_mode(enum pt_exec_mode mode);
static void xed_print_insn(const xed_decoded_inst_t *inst, uint64_t ip);
static xed_machine_mode_enum_t translate_mode(enum pt_exec_mode mode);
static void print_raw_insn(const struct pt_insn *insn);
static void print_raw_insn_file(const struct pt_insn *insn);
static int drain_events_insn(struct pt_insn_decoder *decoder, int status);

static const char *print_exec_mode(enum pt_exec_mode mode)
{
  switch (mode)
  {
  case ptem_unknown:
    return "<unknown>";

  case ptem_16bit:
    return "16-bit";

  case ptem_32bit:
    return "32-bit";

  case ptem_64bit:
    return "64-bit";
```

```c
  }

  return "<invalid>";
}

static void print_raw_insn(const struct pt_insn *insn)
{
  uint8_t length, idx;
  if (!insn)
  {
    printf("[internal error]");
    return;
  }
  printf("             ");
  length = insn->size;
  if (sizeof(insn->raw) < length)
    length = sizeof(insn->raw);

  for (idx = 0; idx < length; ++idx)
    printf("%02x", insn->raw[idx]);

  for (; idx < pt_max_insn_size; ++idx)
    printf("   ");
}

static void print_raw_insn_file(const struct pt_insn *insn)
{
  uint8_t length, idx;

  if (!insn)
  {
    printf("[internal error]");
    return;
  }

  length = insn->size;
  if (sizeof(insn->raw) < length)
    length = sizeof(insn->raw);

  for (idx = 0; idx < length; ++idx)
    fprintf(bufferFd, "%02x", insn->raw[idx]);

  for (; idx < pt_max_insn_size; ++idx)
    fprintf(bufferFd, "   ");
  fprintf(bufferFd, "\n");
}

static int drain_events_insn(struct pt_insn_decoder *decoder, int status)
{
  int errcode;
  while (status & pts_event_pending)
  {
    struct pt_event event;
    uint64_t offset;

    status = pt_insn_event(decoder, &event, sizeof(event));
    if (status < 0)
```

```
      return status;
  }

  return status;
}

static void xed_print_insn(const xed_decoded_inst_t *inst, uint64_t ip)
{
  xed_print_info_t pi;
  char buffer[256];
  xed_bool_t ok;

  if (!inst)
  {
    printf(" [internal error]");
    return;
  }

  // Print raw instruction
  /*
  xed_uint_t length, i;

  length = xed_decoded_inst_get_length(inst);
  for (i = 0; i < length; ++i)
    printf(" %02x", xed_decoded_inst_get_byte(inst, i));

  for (; i < pt_max_insn_size; ++i)
    printf("   ");
  */

  xed_init_print_info(&pi);
  pi.p = inst;
  pi.buf = buffer;
  pi.blen = sizeof(buffer);
  pi.runtime_address = ip;

  // AT&T syntax
  // pi.syntax = XED_SYNTAX_ATT;

  ok = xed_format_generic(&pi);
  if (!ok)
  {
    printf(" [xed print error]");
    return;
  }

  printf(" %s ", buffer);
}

/*
Identifies processor instruction set mode that we are decoding
*/
static xed_machine_mode_enum_t translate_mode(enum pt_exec_mode mode)
{
  switch (mode)
  {
  case ptem_unknown:
```

```c
    return XED_MACHINE_MODE_INVALID;

  case ptem_16bit:
    return XED_MACHINE_MODE_LEGACY_16;

  case ptem_32bit:
    return XED_MACHINE_MODE_LEGACY_32;

  case ptem_64bit:
    return XED_MACHINE_MODE_LONG_64;
  }
  return XED_MACHINE_MODE_INVALID;
}

static void print_insn(const struct pt_insn *insn, xed_state_t *xed, uint64_t offset)
{
  if (!insn)
  {
    printf("[internal error]\n");
    return;
  }

  print_exec_mode(insn->mode);
  // printf("%016" PRIx64 " ", offset);

  printf("%016" PRIx64, insn->ip);

  xed_machine_mode_enum_t mode;
  xed_decoded_inst_t inst;
  xed_error_enum_t errcode;

  mode = translate_mode(insn->mode);

  xed_state_set_machine_mode(xed, mode);
  xed_decoded_inst_zero_set_mode(&inst, xed);

  errcode = xed_decode(&inst, insn->raw, insn->size);
  switch (errcode)
  {
  case XED_ERROR_NONE:
    xed_print_insn(&inst, insn->ip);
    break;

  default:
    print_raw_insn(insn);

    printf(" [xed decode error: (%u) %s]", errcode,
        xed_error_enum_t2str(errcode));
    break;
  }

  printf("\n");
}
```

A.6 binTest.c

```
// C code stored in geeks.c file
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <stdlib.h>

FILE *fptr;
char buffer1[1000];

//sudo gcc -static -no-pie -fno-stack-protector ./test1.c -o bin1.out

void vulnerableFunc(char* input) {
    char buffer[20];
    memcpy(&buffer,input,1000);
}

// Driver Code
int main()
{


    if ((fptr = fopen("./file1.in","r")) == NULL){
       printf("Error!␣opening␣file");
       exit(1);
    }

    char ch;
    int i=0;
    /*
    do {
        ch = fgetc(fptr);
        buffer1[i]=ch;
        i++;
    } while (ch != EOF);
    fclose(fptr);*/
    fread(buffer1, sizeof(char), 1000, fptr);
    fclose(fptr);

    vulnerableFunc(buffer1);
    return 0;
}
```