ARCHITECTURE-INDEPENDENT DYNAMIC INFORMATION FLOW TRACKING

A Thesis Presented

by

Ryan Whelan

to

The Department of Electrical and Computer Engineering

in partial fulfillment of the requirements

for the degree of

Master of Science

in

Electrical and Computer Engineering

Northeastern University

Boston, Massachusetts

November 2012

© Copyright 2012 by Ryan Whelan

All Rights Reserved

Abstract

Dynamic information flow tracking is a well-known dynamic software analysis technique with a wide variety of applications that range from making systems more secure, to helping developers and analysts better understand the code that systems are executing. Traditionally, the fine-grained analysis capabilities that are desired for the class of these systems which operate at the binary level require that these analyses are tightly coupled to a specific ISA. This fact places a heavy burden on developers of these systems since significant domain knowledge is required to support each ISA, and our ability to amortize the effort expended on one ISA implementation cannot be leveraged to support the next ISA. Further, the correctness of the system must carefully evaluated for each new ISA.

In this thesis, we present a *general* approach to information flow tracking that allows us to support multiple ISAs without mastering the intricate details of each ISA we support, and without extensive verification. Our approach leverages binary translation to an intermediate representation where we have developed detailed, architecture-neutral information flow models. To support advanced instructions that are typically implemented in C code in binary translators, we also present a combined static/dynamic analysis that allows us to accurately and automatically support these instructions. We demonstrate the utility of our system in three different application

settings: enforcing information flow policies, classifying algorithms by information flow properties, and characterizing types of programs which may exhibit excessive information flow in an information flow tracking system.

Acknowledgements

First of all, I would like to thank my family for always being there for me, and supporting my endeavors.

Secondly, I would like to thank my advisor, Professor David Kaeli, as well as Professor Engin Kirda. Professor Kaeli's knowledge about computer architecture and security has inspired me to pursue the field with a passion, and his research direction has helped me realize this passion over the past several years. Professor Kirda's expertise in system security was extremely valuable as well.

Lastly, I would like to thank the Cyber System Assessments Group at MIT Lincoln Laboratory for sponsoring this work. This includes Tim Leek for the invaluable mentorship, Michael Zhivich for the meaningful discussions about the work, as well as the other members of the group.

Of course, none of this work would have been possible without the open source software communities that have developed the systems upon which this thesis is based.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Introduction

The development of secure systems is a challenging task with today's popular programming languages, runtime systems, operating systems, and processor architectures. As a result of this, vulnerabilities are perpetually being discovered, exploited, and subsequently patched. These vulnerabilities have provided opportunities for malicious software to proliferate, leading to widespread attacks on consumers, corporations, and governments all over the world.

Figure 1.1 shows just how dismal the state of system security is. In 2012 alone, nearly 5,000 vulnerabilities have been reported to NIST's National Vulnerability Database. While this is better than 2006 when nearly 7,000 vulnerabilities were reported, there is still a great deal of work to be done to ensure secure system development.

The many thousands of vulnerabilities reported in recent years has given way to many millions of new malware samples appearing each year. Over 20 million new

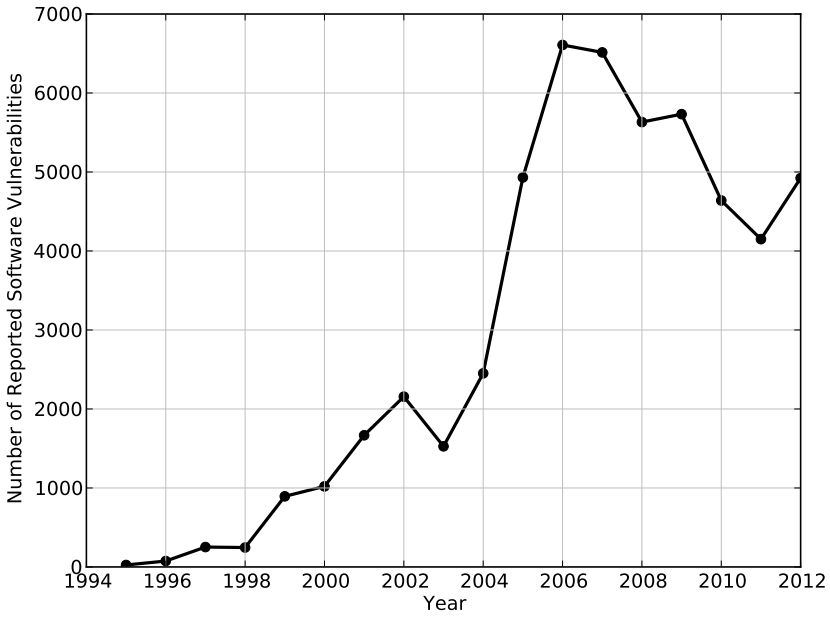


Figure 1.1: Number of vulnerabilities reported to the National Vulnerability Database per year [3]

malware samples have appeared in 2012, and Figure 1.2 shows additional data from the previous two years from the McAfee Quarterly Threat Report. Additionally, over 20,000 of the malware samples from 2012 were specifically developed for mobile phones [2]. This wide landscape of vulnerabilities and malware is why thousands of serious attacks and data breaches happen every year.

To combat the problems of vulnerabilities and malware, many software analysis techniques have been proposed and developed. These techniques are either *static*, where source or binary code is analyzed without actually executing it, or *dynamic*, where analysis is performed as code is executing. For example, static source code analysis can be integrated into development environments and attempt to alert developers of potential vulnerabilities during development time. Static binary analysis

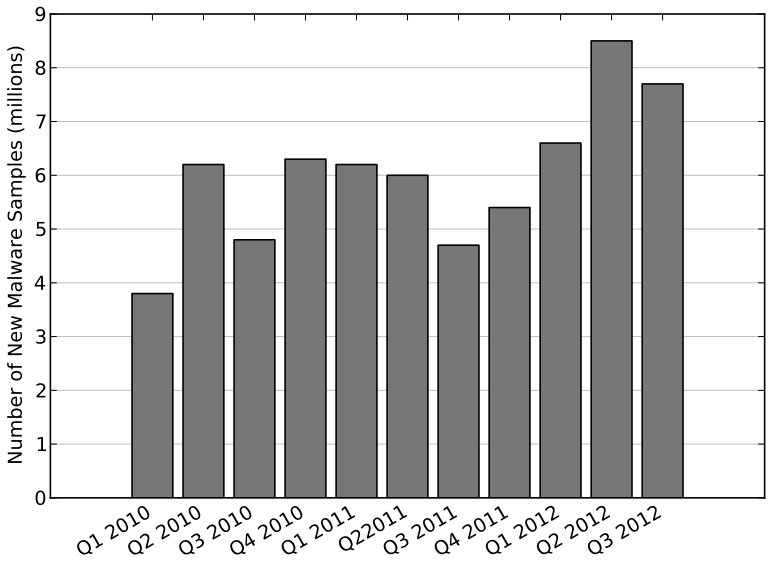


Figure 1.2: Number of new malware samples in the McAfee Quarterly Threat Report [2]

frameworks can inspect malware to attempt to learn as much about the behavior of each sample as possible. Dynamic techniques can both detect the kinds of programming errors that can indicate serious vulnerabilities, and monitor malware as it is executing in order to develop behavioral profiles of each sample.

One method of dynamic software analysis is dynamic information flow tracking, also known as dynamic taint analysis. This is a well-known software analysis technique that has been shown to have wide applicability in software analysis and security applications. However, since dynamic information flow tracking systems that operate at the binary level require fine-grained analysis capabilities to be effective, this means that they are generally tightly coupled with the ISA of code to be analyzed.

In order to implement a fine-grained analysis capability such as information flow

tracking for the ISA of interest, an intimate knowledge of the ISA is required in order to accurately capture information flow for each instruction. This is especially cumbersome for ISAs with many hundreds of instructions that have complex and subtle semantics (e.g., x86). Additionally, after expending the work required to complete such a system, the implementation only supports the single ISA, and a similar effort is required for each additional ISA. To overcome this challenge, we've elected to take a compiler-based approach by translating architecture-specific code into an architecture-independent intermediate representation where we can develop, reuse, and extend a single set of analyses.

In this thesis, we present PIRATE: a Platform for Intermediate Representation-based Analyses of Tainted Execution. PIRATE decouples the tight bond between the ISA of code under analysis and the additional instrumentation code, and provides a general taint analysis framework that can be applied to a large number of ISAs. PIRATE leverages QEMU [7] for binary translation, and LLVM [22] as an intermediate representation within which we can perform our architecture-independent analyses. We show that our approach is both *general* enough to be applied to multiple ISAs, and *precise* enough to provide the detailed kind of information expected from a fine-grained dynamic information flow tracking system. In our approach, we define detailed byte-level information flow models for 29 instructions in the LLVM intermediate representation which gives us coverage of thousands of instructions that appear in translated guest code. We also apply these models to complex guest instructions that are implemented in C code. To the best of our knowledge, this is the first implementation of a binary level dynamic information flow tracking system that is general enough to be applied to multiple ISAs without requiring source code.

1.2 Contributions

The contributions of this work are:

- A framework that leverages dynamic binary translation producing LLVM intermediate representation that enables architecture-independent dynamic analyses, and a language for precisely expressing information flow of this IR at the byte level.
- A combined static/dynamic analysis to be applied to the C code of the binary translator for complex ISA-specific instructions that do not fit within the IR, enabling the automated analysis and inclusion of these instructions in our framework.
- An evaluation of our information flow tracking framework for x86, x86_64, and ARM, highlighting three security-related applications: 1) enforcing information flow policies, 2) characterizing algorithms by information flow, and 3) diagnosing sources of state explosion for each ISA.

1.3 Organization of the Thesis

The rest of this thesis is organized as follows. In Chapter 2, we review work related to this thesis in several different areas. In Chapter 3, we introduce the concept of information flow tracking in detail. Chapters 4 and 5 present the architectural overview and implementation of Pirate. Chapter 6 presents our evaluation with three security-related applications, while Chapter 7 includes some additional discussion. We then conclude in Chapter 8.

Chapter 2

Related Work

We discuss related work in the following sections under these areas: binary translation and instrumentation, intermediate representations, and dynamic information flow tracking.

2.1 Binary Translation and Instrumentation

Historically, binary translation was used to enable binary compatibility between more common ISAs (such as x86 and PowerPC) and newer ISAs employing VLIW instructions. For instance, DAISY [17] presents a VLIW architecture designed for emulating the PowerPC ISA. Additionally, FX!32 [12] allows x86 Windows applications to run on the Alpha ISA, while the Transmeta Crusoe [16] allows x86 code to run on a VLIW ISA. All of these systems take advantage of the translation step to also dynamically optimize code, while BOA [5] is explicitly focused on translating PowerPC to VLIW for aggressive optimization. More recently, QEMU [7] has proven to be a robust binary translator with runtime support for both programs and entire operating systems, and translation support for 14 guest ISAs and 9 host ISAs.

While the translation step in binary translators provides the opportunity to dynamically optimize code, it also presents the opportunity to dynamically instrument code without requiring the original source. ATOM [35] presents a general framework for adding instrumentation code to binaries at link time. DynamoRIO [8], Valgrind [29], and Pin [24] are similar to ATOM, except they have the advantage of operating on binaries at run time, eliminating the need for a customized linker. These tools present useful APIs enabling a wide variety of binary instrumentation tools to be developed. In fact, many of the security tools described in Section 2.3 rely on dynamic binary instrumentation, and are built with tools such as QEMU, DynamoRIO, Valgrind, and Pin.

2.2 Intermediate Representations

Intermediate representations (IRs) are low-level, machine-independent languages that compilers use to analyze and optimize code in a retargetable way. IRs allow compilers that support compilation to multiple ISAs to maintain a single set of optimizations, rather than multiple optimizations duplicated for each supported ISA. As an example, the intermediate representation of the LLVM compiler infrastructure [22] is robust from both the language perspective on the front end of the compiler, and the ISA perspective on the back end of the compiler. Consequently, LLVM can support many languages (C, Objective C, C++, OpenCL, Python, Ruby, Haskell, etc.) and ISAs (x86, x86_64, ARM, MIPS, PowerPC, SPARC, etc.) with a mature set of analyses and optimizations developed in terms of the IR. QEMU uses an IR during the binary translation process that allows it to apply a single set of optimizations to each supported ISA.

IRs have been shown to be useful not only in compilers and binary translators, but also in software analyses. In particular, we leverage the LLVM IR in our analysis in order to achieve architecture independence. Valgrind [29] employs an intermediate representation, but it is limited to user-level programs which would prevent us from extending our work to entire operating systems. The CMU Binary Analysis Platform (BAP) [9] defines an intermediate representation for software analysis, but that system currently can only analyze x86 and ARM programs, and it doesn't have x86_64 support. CodeSurfer/x86 [31] shows how x86 binaries can be statically lifted to an intermediate representation enabling various static analyses on x86 binaries.

2.3 Dynamic Information Flow Tracking

Dynamic information flow tracking has been shown to have a wide variety of real world applications, including the detection and prevention of exploits for binary programs [11, 30, 42] and web applications [38]. Applications to malware analysis include botnet protocol reverse engineering [10, 41], and identifying cryptographic primitives in malware [25]. For debugging and testing assistance, dynamic information flow tracking can be used to visualize where information flows in complex systems [26], or to improve code coverage during testing [23]. Additionally, this technique can be used for automated software vulnerability discovery [19, 40].

Information flow tracking has been implemented in a variety of ways, including at the hardware level [37, 39], in software through the use of source-level instrumentation [19, 23, 42], binary instrumentation [15, 21, 32], or the use of a whole-system emulator [14, 30, 34]. Additionally, it can also be implemented at the Java VM

layer [18]. Between all of these different implementations, the most practical approach for analyzing real-world software (malicious and benign) when source code is not available is to perform information flow tracking at the binary level. PIRATE is the first information flow tracking system that works at the binary level that supports multiple ISAs that can trivially be extended to at least a dozen ISAs.

Existing systems that are the most similar to PIRATE are Argos [30], BitBlaze [34], Dytan [15], and Libdft [21]. These systems have contributed to significant results in the area of dynamic information flow tracking. Argos and BitBlaze are implemented with QEMU [7], while Dytan and Libdft are implemented with Pin [24]. Even though QEMU and Pin support multiple guest ISAs, each of these information flow tracking systems are tightly coupled with x86, limiting their applicability to other ISAs.

In looking at all of the aforementioned related work, we decided that the most flexible system would consist of a combination of QEMU and LLVM. QEMU's support of 14 different guest architectures, and the flexibility of translating programs or entire operating systems gives us a method to support a wide variety of systems. LLVM's extensible compiler framework and rich intermediate representation provide us with an ideal environment to implement our system.

Chapter 3

Dynamic Information Flow Tracking

In this chapter, we present dynamic information flow tracking in detail – using a shadow memory to keep track of every byte in memory, different methods of performing information flow tracking, and some of the challenges of using this technique for dynamic software analysis.

3.1 Shadow Memory

The fundamental idea behind information flow tracking is that data of interest is labeled and subsequently tracked as it propagates through a system. In the most general case, data that is labeled is not trusted, so existing systems often implement policies that guard against dangerous operations on untrusted (or 'tainted') data. The key component in the architecture of dynamic information flow tracking systems that allows data to be tracked at a fine-grained level is a *shadow memory* [28, 43].

Using a shadow memory allows every unit of data to be tracked as it flows through

the system. Generally data is labeled and tracked at the byte level, but this can also happen at the bit, word, or even page level, depending on the desired granularity. The shadow memory can also store data at varying granularities. In the simplest case, each unit of data is also accompanied by one bit of data (tracked or not tracked). More intricate shadow memories track one byte of data (a small number), or a data structure that tracks additional information. Tracking additional information is useful for the cases when *label sets* are propagated through the system.

In order to propagate the flow of data, major components of the system need to be shadowed to keep track of where tagged data flows. This includes CPU registers, memory, and in the case of whole systems, the hard drive also. When information flow tracking is implemented for binaries at a fine-grained level, this means that propagation occurs at the level of the ISA where single instructions that result in a flow of information are instrumented. This instrumentation updates the shadow memory accordingly when tagged information is propagated through instructions that transfer data.

3.2 Methods of Tracking Information Flow Through Systems

Hardware-based. Information flow tracking can occur at many different levels in the hardware/software stack. The lowest level to perform this analysis is directly within the hardware itself [11, 37, 39]. Hardware-based implementations generally

provide the lowest overhead, but require significant architectural and microarchitectural changes to the processor. These limitations mean that hardware-based approaches are the least flexible, and also the least practical since commonly used processor architectures weren't designed with this capability.

Source-code based. Information flow tracking can also be performed at the source code level through the use of source-level instrumentation [19, 23, 42]. Systems like these use source-to-source transformation with frameworks like CIL [27] to automatically insert the necessary instrumentation code to propagate tags. While this approach is both fast and flexible, it is less practical since it requires source code to be available.

Binary instrumentation. To perform information flow tracking without requiring hardware modifications or available source code, other platforms perform instrumentation of binaries [15, 21, 32] or instrumentation of whole-system emulators [14, 30, 34]. This approach enables a wide variety of analyses to be developed for both malicious and benign code, for user-level programs and entire operating systems. Despite the flexibility of this approach, it generally has high overheads since instrumentation code needs to be executed for each guest instruction. For this thesis, this approach is the one we are most interested in.

3.3 Challenges

Under-tainting due to implicit flows. In terms of information flow tracking, many instructions operate on data in such a way that information flow is explicit. For example, memory operations transfer data to and from memory, and arithmetic

operations transfer data from the operand(s) to the destination. Tag propagation for these types of instructions is generally straightforward, and no tag information is lost. The problem of under-tainting presents itself when control flow decisions based on tagged data affect other data that is used in the program. One example of this is plaintext-to-RTF conversion [20]. In this conversion process, control flow decisions are based on input data, and this directly affects data that becomes part of the output. Because of the separate data and control dependencies in this situation, most information flow tracking systems will not propagate tags to the output data. This problem makes it possible for people (such as malware authors) to deceive information flow tracking systems. While we don't address this problem in this thesis, Kang et al. [20] present a promising solution.

Over-tainting due to state explosion. On the contrary, over-tainting can also present a problem for information flow tracking systems [33]. The problem of state explosion presents itself when an excessive amount of tagged data is being tracked through the system, when in fact only a small subset of that data is the data of interest to be tracked. Often, state explosion is a side effect of tracking pointers in the same way as data. Tagged pointer tracking can sometimes be responsible for state explosion, but it is necessary in order to detect keyloggers [33], non-control data attacks [11] and to properly track information flow with the use of structures such as lookup tables. We provide a characterization of state explosion for several pointer-intensive programs in Section 6.3.

Tight coupling to a specific ISA. Existing dynamic information flow tracking systems that employ binary instrumentation are inherently coupled to the ISA of code being analyzed. Due to the popularity of the x86 ISA, and the tight bond of

these binary instrumentation techniques with the ISA under analysis, most of these systems have been carefully designed to correctly propagate information flow only for the instructions that are included in x86. This imposes a significant limitation on dynamic information flow tracking, since a significant effort is required to support additional ISAs. PIRATE solves this problem by decoupling this analysis technique from the underlying ISA, without requiring source code or higher-level semantics.

Chapter 4

PIRATE Overview

At a high level, PIRATE works as follows. At the core is the QEMU binary translator [7]. QEMU is a versatile dynamic binary translation platform that can translate 14 different guest architectures to 9 different host architectures by using its own custom intermediate representation, using the Tiny Code Generator (TCG). In our approach, we take advantage of this translation to IR to support information flow tracking for multiple guest architectures. However, the TCG IR consists of very simple RISC-like instructions, making it difficult to represent complex ISA-specific instructions. To overcome this limitation, the QEMU authors implement a large number of guest instructions in *helper functions*, which are C implementations of these instructions. Each guest ISA implements hundreds of instructions in helper functions that perform critical computations, so we have devised a mechanism to automatically track information flow to, from, and within these helper functions.

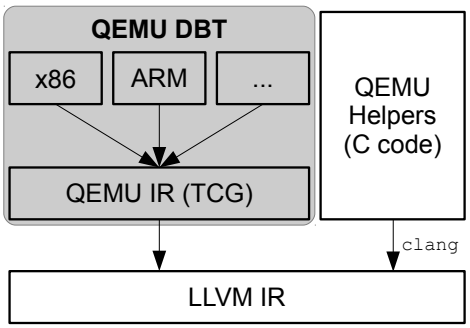


Figure 4.1: Lowering code to the LLVM IR with QEMU and Clang

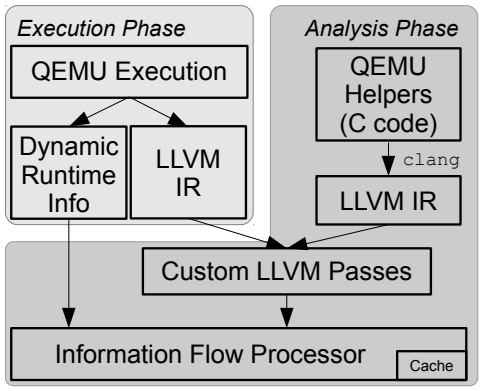


Figure 4.2: System architecture split between execution and analysis

```
void glue(helper_pshufw, SUFFIX) (Reg *d, Reg *s, int order){
    Reg r;
    r.W(0) = s->W(order & 3);
    r.W(1) = s->W((order >> 2) & 3);
    r.W(2) = s->W((order >> 4) & 3);
    r.W(3) = s->W((order >> 6) & 3);
    *d = r;
}
```

Figure 4.3: QEMU helper function implementing the `pshufw` (packed shuffle word) MMX instruction for x86 and x86_64 ISAs

4.1 Execution and Analysis

We perform dynamic information flow tracking on Linux binaries using the following approach, which is split between execution and analysis. In the execution phase, we run the program as we normally would under QEMU. We have augmented the translation process to add an additional translation step, which translates the TCG IR to the LLVM IR and then executes on the LLVM Just-in-Time (JIT) compiler. This translation occurs at the level of basic blocks of guest code. In Figure 4.1, we show the process of lowering guest code and helper function code to the LLVM IR to be used in our analysis.

In the analysis phase, we have all the information we need to reconstruct the execution and perform detailed information flow tracking at the byte level. Figure 4.2 shows the architecture of our system. Once we have the execution of guest code captured in the LLVM IR, we can perform our analysis over each basic block of guest code with custom IR passes we have developed within the LLVM infrastructure. The first part of our analysis is to derive information flow operations to be executed on our abstract information flow processor. This is an automated process that emits information flow operations that we've specified for each LLVM instruction. After deriving the sequence of information flow operations, we send them through the information flow processor to propagate tags and update the shadow memory. This allows us to keep our shadow memory updated on the level of a guest basic block. When we encounter system calls during our processing, we treat I/O-related calls as sources or sinks of information flow. For example, the `read()` system call is a source that we begin tracking information flow on, and a `write()` system call is a sink where we may want to be notified if tagged information is written to disk.

4.2 Optimizations

QEMU has a *translation block cache* mechanism that allows it to keep a copy of translated guest code, avoiding the overhead of re-translating frequently executed code repeatedly. This optimization permeates to our analysis phase; a guest basic block that is cached may also be executed repeatedly in the LLVM JIT, but it only appears once in the LLVM module that we analyze. This optimization in QEMU also provides us the opportunity to cache information flow operations. As we analyze translated guest code in the representation of basic blocks that may be cached, we can

also perform the derivation of information flow tracking operations once, and then cache; we refer to these as *taint basic blocks*. Once we derive a taint basic block for a guest basic block of code, we deposit it into our *taint basic block cache*.

4.3 Static Analysis of QEMU Helper Functions

Since QEMU helper functions perform critical computations on behalf of the guest, we need to include them in our analysis as well. To do this, we use Clang [1] to translate helper functions to the LLVM IR. From there, we can use the exact same analysis that we apply to translated guest code to derive information flow operations. Since this is a static analysis, we can emit the corresponding information flow operations for each helper function into a *persistent cache*, which saves us the overhead of invoking Clang and performing our static analysis at the same time that we are performing information flow tracking. Figure 4.3 shows the helper function that implements the `pshufw` MMX instruction. Automatically analyzing functions like these takes a significant burden off of developers of information flow tracking systems.

Chapter 5

Implementation

In this chapter, we present implementation details of PIRATE. The implementation consists of several major components; the execution and trace collection engine, the information flow processor which propagates tagged data, the shadow memory which maintains tagged data, the analysis engine which performs analysis and instrumentation passes over the LLVM IR, and the caching mechanism that caches information flow operations and reduces overhead. Currently, we support information flow tracking for the x86, x86_64, and ARM ISAs on Linux binaries, but our approach trivially extends to other ISAs that are supported by QEMU. Our system is implemented with QEMU 1.0.1 and LLVM 3.0.

5.1 Dynamic Binary Translation to LLVM IR

At the core of our approach is the translation of guest code to an ISA-neutral IR. Much like a standard compiler, we want to perform our analyses in terms of an IR, which allows us to decouple from the ISA-specific details. We take advantage of the fact that QEMU's dynamic binary translation mechanism translates guest code to its

own custom IR (TCG), but this IR is not robust enough for our analyses. In order to bridge the gap between guest code translated to the TCG IR and helper functions implemented in C, we chose to perform our analysis in the LLVM IR. Since both the TCG and LLVM IR consist of simple RISC-like instructions, we have a straightforward translation from TCG to LLVM. For this translation, we leverage the TCG to LLVM translation module included as part of the S2E framework [13]. LLVM also enables us to easily translate helper functions to its IR (through the Clang front-end), and it provides a rich set of APIs for us to work with. By performing information flow tracking in the LLVM IR, we abstract away the intricate details of each of our target ISAs, leaving us with less than 50 RISC-like instructions that we need to understand in great detail and model correctly for information flow analysis.

Developing detailed information flow tracking models for this small set of LLVM instructions that are semantically equivalent to guest code means that our information flow tracking will also be semantically equivalent. This also gives a degree of assurance about the completeness and correctness of our approach.

5.2 Decoupled Execution and Analysis

In PIRATE, we decouple the execution and analysis of code in order to give us flexibility in altering our analyses on a single execution. We capture a compact dynamic trace of the execution in the LLVM bitcode format, along with dynamic values from the execution that include memory access addresses, and branch targets. We obtain these dynamic values by instrumenting the IR to log every address of loads and stores, and every branch taken during execution. The code we capture is in the format of

an LLVM bitcode module which consists of a series of LLVM functions, each corresponding to a basic block of guest code. We also capture the order in which these functions are executed. Our trace is compact in the sense that if a basic block is executed multiple times, we only need to capture it once.

Once we've captured an execution, we leverage the LLVM infrastructure to perform our analysis directly on the LLVM IR. Our analysis is applied in the form of an LLVM analysis pass, where we specify the set of *information flow operations* for each LLVM instruction in the execution. We perform this analysis at the granularity of a guest basic block, and our analysis emits a *taint basic block*. Our abstract information flow processor then processes these taint basic blocks to update the shadow memory accordingly.

5.3 Shadow Memory, Information Flow Processor

In order to propagate information flow through a process, we use a shadow memory to keep track of information flow at the byte level. Information flow is propagated through *information flow operations* that are executed on our abstract *information flow processor*. Information flow operations propagate tagged data at the byte level, and the shadow memory is updated at the end of each guest basic block.

Shadow Memory. Our shadow memory keeps track of information flow of all data in a process at the byte level. It is partitioned into the following segments: virtual memory, architected registers, and LLVM registers (which include multiple calling scopes). The virtual memory portion of the shadow memory keeps track of information flow through the process based on virtual addresses. The architected state

portion keeps track of general purpose registers, and also some special purpose registers (such as MMX registers for x86). The LLVM shadow registers are how we keep track of information flow between LLVM IR instructions. LLVM is an SSA-based IR with an infinite amount of abstract registers. This means that every new value that gets defined in an LLVM function is assigned to a new abstract register within that function. Currently, our shadow memory models 2,000 abstract registers. We maintain multiple scopes of abstract LLVM registers in our shadow memory to accomodate the calling of helper functions, which are explained in more detail in Section 5.5. The shadow memory is configurable so data can be tracked at the binary level (tagged or untagged), or positionally with multiple labels per address, which we refer to as a *label set*. Since we are modeling the entire address space of a process in our shadow memory, it is important that we utilize an efficient implementation. For 32-bit ISAs, our shadow memory of the virtual address space consists of a two-level structure that maps a directory to tables with tables that map to pages, similar to x86 virtual addressing. For 64-bit ISAs, we instead use a five-level structure in order to accommodate the entire 64-bit address space. To save memory overhead, we only need to allocate shadow guest memory for memory pages that contain tagged information.

Deriving Information Flow Operations. On our abstract information flow processor, we execute information flow operations in order to propagate tags and update the shadow memory. These operations specify information flow at the byte level. An address can be a byte in memory, a byte in an architected register, or a byte in an LLVM abstract register. The set of information flow operations can be seen in Table 5.1. Here, we describe them in more detail.

Operation	Semantics
$label(a, l)$	$L(a) \leftarrow L(a) \cup l$
$delete(a)$	$L(a) \leftarrow \emptyset$
$copy(a, b)$	$L(b) \leftarrow L(a)$
$compute(a, b, c)$	$L(c) \leftarrow L(a) \cup L(b)$
$insn\_start$	Bookkeeping info
$call$	Begin processing a QEMU helper function
$return$	Return from processing a QEMU helper function

Table 5.1: Information flow operations

- `label:` Associate label l with the set of labels that belong to address a.
- `delete:` Discard the label set associated with address a.
- `copy:` Copy the label set associated with address a to address b.
- `compute:` Address c gets the union of the label sets associated with address a and address b.
- `insn_start:` Maintains dynamic information for operations. For loads and stores, a value from the dynamic log is filled in. For branches, a value from dynamic log is read to see which branch was taken, and which basic block of operations needs to be processed next.
- `call:` Indication to process information flow operations for a QEMU helper function. Shift information flow processor from caller scope to callee scope, which has a separate set of shadow LLVM registers. Retrieve information flow operations from the persistent cache. If the helper function takes arguments, propagate information flow of arguments from caller scope to callee scope.

LLVM Instruction:

```
%32 = xor i32 %30, %31;
```

Information Flow Operations:

```
compute(%30[0], %31[0], %32[0]);
compute(%30[1], %31[1], %32[1]);
compute(%30[2], %31[2], %32[2]);
compute(%30[3], %31[3], %32[3]);
```

LLVM Instruction:

```
%7 = load i32* %2;
```

Information Flow Operations:

```
// get load address from dynamic
// log, and fill in next
// four operations
insn_start;

// continue processing operations
copy(addr[0], %7[0]);
copy(addr[1], %7[1]);
copy(addr[2], %7[2]);
copy(addr[3], %7[3]);
```

Figure 5.1: Examples of byte-level information flow operations for 32-bit `xor` and `load` LLVM instructions

- `return`: Indication that processing of a QEMU helper function is finished. Shift information flow processor from callee scope to caller scope. If the helper function returns a value, propagate information flow to shadow return value register.

The information flow models we've developed allow us to derive the sequence of information flow operations for each LLVM function using our LLVM analysis pass. In this pass, we iterate over each LLVM instruction and populate a buffer with the corresponding information flow operations. Due to the semantics in the translation, particular functions can generate multiple basic blocks for a single basic block of guest code. In this case, our pass analyzes each basic block in the LLVM function with corresponding *taint basic blocks*. In Figure 5.1, we show sequences of information flow operations for the LLVM `xor` and `load` instructions.

5.4 Caching of Information Flow Tracking Operations

One of the main optimizations that QEMU implements is the *translation block cache* which saves the overhead of retranslating guest code to host code for frequently executed basic blocks. We took a similar approach for our *taint basic blocks* and developed a caching mechanism to eliminate the need to repeatedly derive information flow operations. This means we only need to run our pass once on a basic block, and as long as it is in our cache, we simply process the information flow operations.

Our caching mechanism works as follows. During our analysis pass, we leave dynamic values such as memory accesses and taken branches empty, and instead fill them in at processing time by using our `insn_start` operation, as illustrated in Figure 5.1. In the case of a branch, the `insn_start` operation tells the information flow processor to consult the dynamic log to find which branch was taken, and continue on to process that *taint basic block*. This technique enables us to process cached information flow operations with minor preprocessing to adjust for dynamic information.

5.5 Analysis and Processing of QEMU Helper Functions

Instrumentation and Analysis. Because important computations are carried out in helper functions, we need some mechanism to analyze them in a detailed, correct way. Because there are hundreds of helper functions in QEMU, this process needs to be automated. We have modified the QEMU build process to automatically derive information flow operations for helper functions, and save them to a *persistent cache*.

Here, we describe that process in more detail.

1. **Translate helper function C code to LLVM IR using Clang.**
   The Clang compiler [1], which is a C front-end for the LLVM infrastructure, has an option to emit a LLVM bitcode file for a compilation unit. We have modified the QEMU build process to do this for compilation units which contain helper functions we are interested in analyzing.

2. **Run our LLVM function analysis pass on the helper function LLVM.**
   Once we have helper function code in the LLVM format, we can compute information flow operations using the same LLVM analysis pass that we have developed for use on generated code.

3. **Instrument the LLVM IR to populate the dynamic log.**
   In order for us to perform our analysis on the helper function LLVM, we need this code to populate the dynamic log with load, store, and branch values. We have developed a simple code transformation pass that instruments the helper function IR with this logging functionality.

4. **Emit information flow operations into a persistent cache.**
   Helper function information flow operations can be emitted into a persistent cache because they are static, and because runtime overhead will be reduced by performing these computations at compile time. This cache is now another by-product of the QEMU build process.

5. **Compile and link the instrumented LLVM.**
   Since the instrumented IR should populate the dynamic log during the trace collection, we create an object file that can be linked into QEMU. Again, we can

use Clang to translate our instrumented LLVM bitcode into an object file, and then link that file into the QEMU executable during the QEMU build process.

Processing. Integration of helper functions analysis into our information flow tracking framework works as follows. During analysis of QEMU generated code, we see a call to a helper function, arguments (in terms of LLVM registers, if any), and return value (in terms of a LLVM register, if any). When we see a call instruction in our analysis pass on translated code, we propagate the information flow of the arguments to the callee's scope of LLVM registers, if necessary. For example, assume in the caller's scope that there is a call to `foo()` with values `%29` and `%30` as arguments. In the scope of the helper function, the arguments will be in values `%0` and `%1`. So the information flow of each argument gets copied to the callee's scope, similar to how arguments are passed to a new scope on a stack. We then insert our `call` operation, which tells the information flow processor which function to process, and the pointer to the set of corresponding taint operations that are in the cache. The information flow processor then processes those operations, until return. On return, a helper function may or may not return a value to the previous scope. For return, we emit a `return` operation to indicate that we are returning to the caller's scope. If a value is returned, then its information will be present in the LLVM return value register in our shadow memory so if there are any tags on that value, they will be propagated back to the caller's scope correctly.

Chapter 6

Evaluation

In our evaluation, we show that PIRATE is decoupled from a specific ISA, bringing the utility of information flow tracking to software developers and analysts regardless of the underlying ISA they are targeting. We demonstrate the following three applications for x86, x86_64, and ARM: enforcing information flow policies, algorithm characterization, and state explosion characterization. We performed our evaluation on Ubuntu 64-bit Linux 3.2, and in each case, we compiled programs with GCC 4.6 with default options for each program.

6.1 Enforcing Information Flow Policies

One important application of dynamic information flow tracking is to define information flow policies for applications, and ensure that they are enforced within the application. For example, one may define a policy that a certain subset of program data is not allowed to be sent out over the network, or that user-provided data may not be allowed to be passed to security-sensitive functions. A universal information flow policy that most programs enforce is that user-provided data may not be used to

Program	x86	x86_64	ARM
Hello World	10/104 (9.62%)	11/93 (11.83%)	10/100 (10.00%)
Gzip Compress	17/150 (11.33%)	15/147 (10.20%)	17/150 (11.33%)
Bzip2 Compress	16/167 (9.58%)	16/153 (10.46%)	17/165 (10.30%)
Tar Archive	2/391 (0.51%)	2/372 (0.54%)	2/361 (0.55%)
OpenSSL AES Encrypt	8/674 (1.19%)	7/655 (1.07%)	7/671 (1.04%)
OpenSSL RC4 Encrypt	4/672 (0.59%)	4/653 (0.61%)	5/679 (0.73%)
KL Graph Partition	29/132 (21.97%)	63/122 (51.64%)	32/134 (23.88%)

Table 6.1: Functions which operate on tagged data

overwrite the program counter. However, there is a lack of information flow tracking systems that support embedded systems employing ARM, MIPS, and PowerPC, and even x86_64, so defining and verifying these policies without modifying source code is difficult or impossible with existing information flow tracking systems.

Our system enables software developers to define and enforce these information flow policies, regardless of the ISA they are developing for. In one set of experiments, we carried out a buffer overflow exploit for a vulnerable program and our system was able to tell us exactly which bytes from our input were overwriting the program counter for x86, x86_64, and ARM.

In addition to telling the developer where in the program these information flow policies are violated, Pirate can also tell the developer each function in the program where tagged data flows. This can assist the developer in identifying parts of the program that operate directly on user input so they can more clearly identify where to focus when ensuring the security of their program. In Table 6.1, we present results for the ratio of functions in several programs that operate on tagged data. These ratios indicate the percentage of functions in the program that operate on tagged data compared with every function executed in the dynamic trace. For most of the

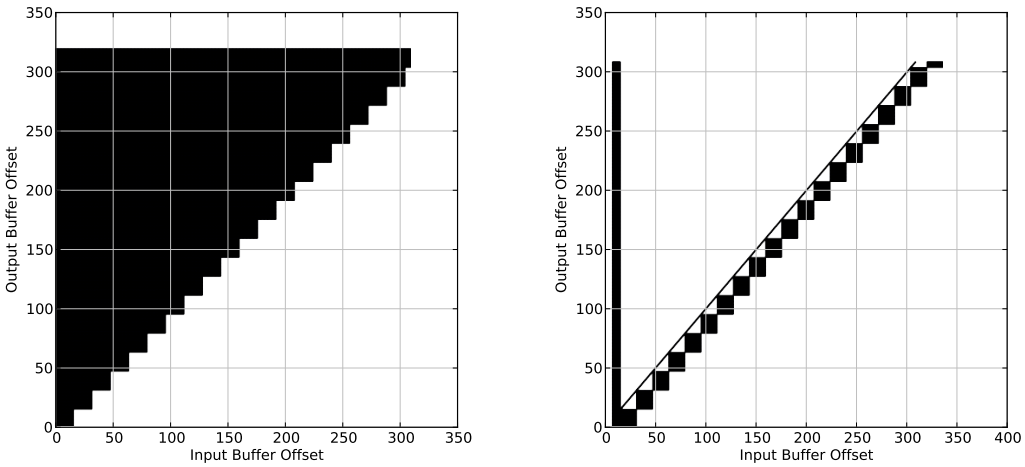
programs we evaluated, these ratios are under 25%. The exception is KL graph partition for x86_64, which shows effects of state explosion. This is addressed in more detail in Section 6.3.

With this enhanced security capability, software developers can more easily identify parts of their programs that may be more prone to attacks. This capability can also be used in the context of vulnerability discovery when source code isn't available. Since dynamic information flow tracking can be effective in finding serious vulnerabilities in programs [19, 40], this technique can also be used in the context of vulnerability discovery in binaries compiled for different ISAs.

6.2 Algorithm Characterization

Recent work has shown that dynamic analysis techniques such as information flow tracking can help malware analysts better understand the obfuscation techniques employed by malware [10, 25]. However, these approaches suffer the same limitations as other systems, where they are tightly-coupled with a single ISA (x86). As embedded systems are becoming increasingly relevant in the security community, it is becoming more desirable for analysts to leverage the power of dynamic information flow tracking for these embedded ISAs.

Here, we highlight the capability of our system to characterize encryption algorithms based on *input/output dependency graphs*. We generate these graphs by positionally labeling each byte in the buffer after the `read()` system call, and tracking how each byte of the input is propagated through the encryption algorithm. By subsequently interposing on the `write()` system call, we can inspect each byte in the buffer to see the set of input bytes that influences each output byte. For these



(a) Encryption (b) Decryption

Figure 6.1: AES CBC mode input/output dependency graphs

experiments, we chose OpenSSL 1.0.1c [4] as a test suite for two modes of AES block cipher encryption, and RC4 stream cipher encryption for x86, x86_64, and ARM. The OpenSSL suite is ideal for demonstrating our capability because most of the encryption algorithms have both C implementations and optimized handwritten assembly implementations.

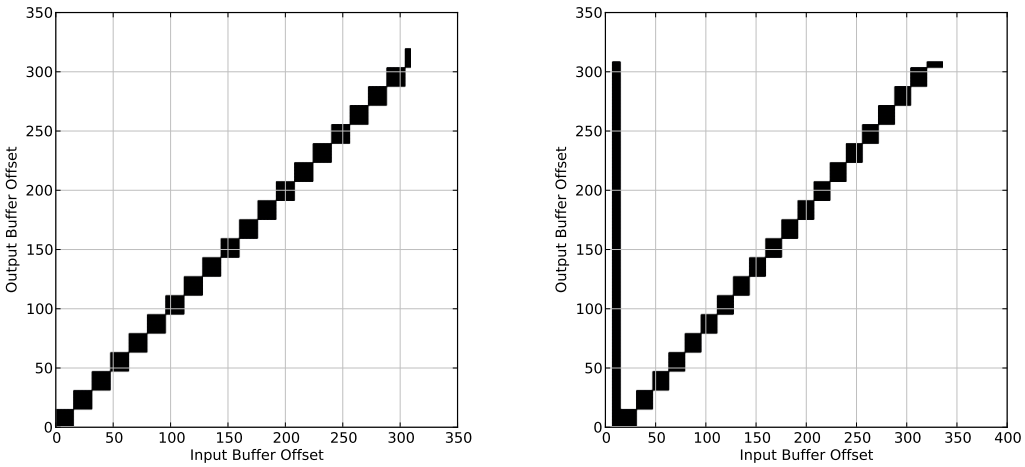
AES, Cipher Block Chaining Mode. AES (Advanced Encryption Standard) is a block encryption algorithm that operates on blocks of 128 bits of data, and allows for key sizes of 128, 192, and 256 bits [36]. As there are a variety of encryption modes for AES, cipher block chaining mode (CBC) is one of the stronger modes. In CBC encryption, a block of plaintext is encrypted, and then the resulting ciphertext is passed through an exclusive-or operation with the subsequent block of plaintext before that plaintext is passed through the block cipher. Inversely, in CBC decryption,

a block of ciphertext is decrypted, and then passed through an exclusive-or operation with the previous block of ciphertext in order to retrieve the plaintext.

Figure 6.1 shows our input/output dependency graphs for AES encryption and decryption. In these figures, we can visualize several main characteristics of the AES CBC cipher: the block size (16 bytes), and the encryption mode. In Figure 6.1(a), the first block of encrypted data is literally displayed as a block indicating complicated dependencies between the first 16 bytes. We see the chaining pattern as each subsequent block depends on all blocks before it in the dependency graph. In Figure 6.1(b), we can see that each value in the output is dependent on the second eight bytes in the input; this corresponds to the salt value, which is an element of randomness that is included as a part of the encrypted file. We can also see the chaining dependency characteristic of CBC decryption, where each block of ciphertext is decrypted, and then passed through an exclusive-or operation with the previous block of ciphertext. This series of exclusive-or operations is manifested as the diagonal line in Figure 6.1(b). With PIRATE, we were able to generate equivalent dependency graphs for x86, x86_64, and ARM, for both handwritten and C implementations.

This result highlights the versatility of our approach based on the wide variety of implementations of AES in OpenSSL. In particular, the x86 handwritten version is implemented using instructions from the MMX SIMD instruction set. Our automated approach for deriving information flow operations for these advanced instructions allows us to support these instructions without the manual effort that other systems require.

AES, Electronic Code Book Mode. Electronic Code Book (ECB) mode is similar to CBC mode, except that it performs block-by-block encryption without the exclusive-or chaining of CBC mode [36]. The input/output dependency graphs we've

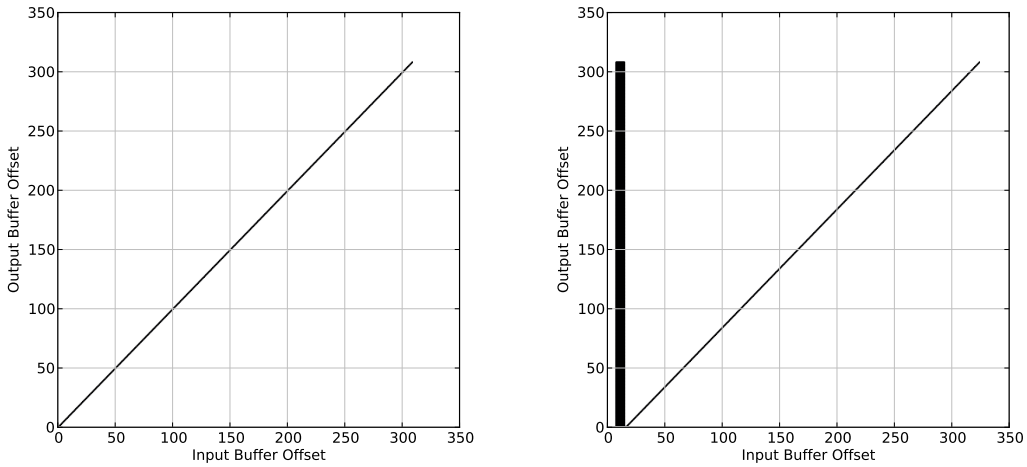


(a) Encryption (b) Decryption

Figure 6.2: AES ECB mode input/output dependency graphs

generated to characterize this algorithm can be seen in Figure 6.2. Here, we see that our system can accurately tell us the block size and the encryption mode, without the chaining dependencies from the previous figures. We again see the dependence on the bytes containing the salt value in Figure 6.2(b).

For AES in ECB mode, we were able to generate equivalent dependency graphs for each ISA (x86, x86_64, and ARM) and implementation (handwritten and C implementation), with the exception of the ARM C implementation for decryption, and the x86 handwritten assembly implementation for encryption. In these exceptional cases, we see a similar input/output dependency graph with some additional apparent data dependence. This highlights a design decision of our system, where we over-approximate information flow transfer of certain LLVM instructions in order to prevent the incorrect loss of tagged information. This over-approximation can manifest itself as additional information flow spread, but we've made the decision that it



(a) Encryption (b) Decryption

Figure 6.3: RC4 input/output dependency graphs

is better to be conservative rather than miss an exploit, especially in the context of security-critical applications of this system.

RC4. RC4 is a stream cipher where encryption occurs through byte-by-byte exclusive-or operations. The algorithm maintains a 256 byte state that is initialized by the symmetric key [36]. Throughout the encryption, bytes in the state are swapped pseudo-randomly to derive the next byte of the state to be passed through an exclusive-or operation with the next byte of the plaintext.

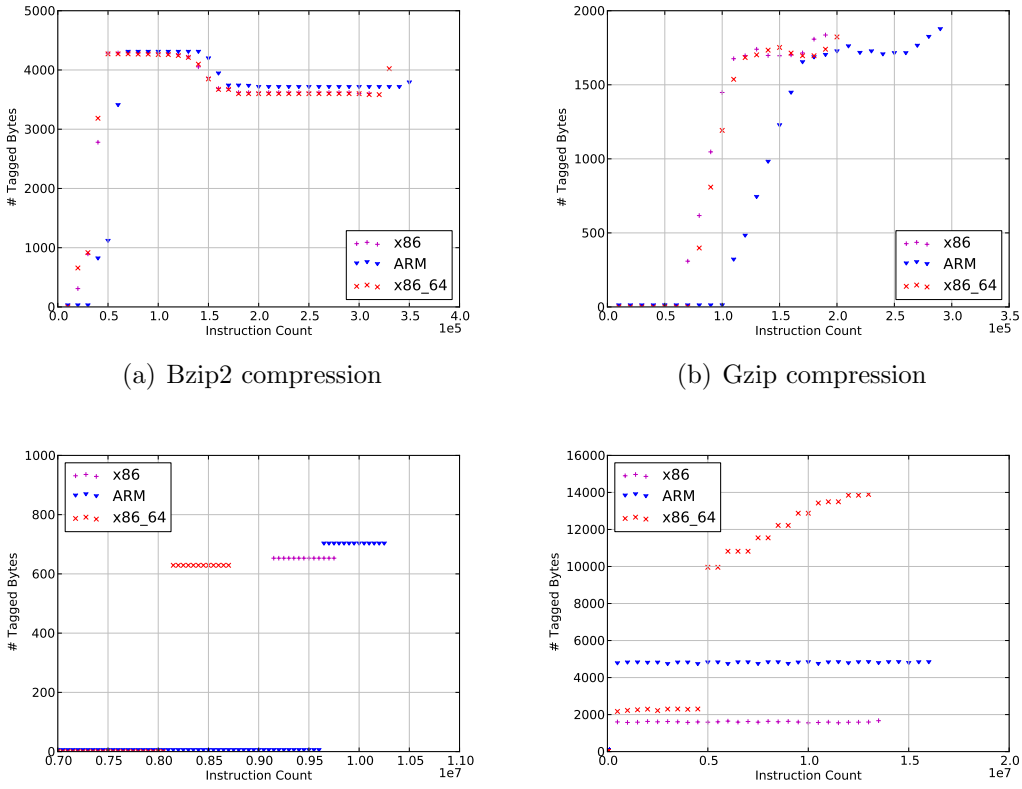
The input/output dependency graphs for RC4 encryption can be seen in Figure 6.3. Since we only track information flow of the input data and not the key, we can see from these figures that there is a linear dependence from input to output, based on the series of exclusive-or operations that occur for each byte in the file. As with the previous figures for decryption, we can see the dependence on the salt value that

is in the beginning of the encrypted file in Figure 6.3(b). For RC4 encryption and decryption, we were able to generate equivalent dependency graphs for encryption and decryption for each ISA (x86, x86_64, and ARM) and implementation (handwritten and C implementation) with the exception of x86_64 encryption and decryption handwritten implementations. For these cases, we see an equivalent dependency with additional information, again due to the conservative approach we take in terms of information flow tracking.

6.3 Diagnosing State Explosion

One limitation of information flow tracking systems is that they are subject to state explosion where tagged data spreads (i.e., grows) uncontrollably, increasing the amount of data that needs to be tracked as it flows through the system. This is especially true when pointers are tracked in the same way as data [33]. Despite this limitation, it is necessary to track pointers to detect and prevent certain kinds of attacks, such as those involved in read or write overflows but where no control flow divergence occurs [11], or those that log keyboard input which is passed through lookup tables [14]. In Pirate, we've implemented tagged pointer tracking as a configurable option. When this option is turned on, we propagate information for loads and stores not only from the addresses that are accessed, but also from the values that have been used to calculate those addresses. Our system allows us to evaluate the effects of state explosion between CISC and RISC ISAs since we support x86, x86_64, and ARM. It also allows us to evaluate the rate of state explosion for different software implementations of the same application.

To perform this evaluation, we've experimentally measured the amount of tagged



(a) Bzip2 compression

(b) Gzip compression

(c) AES CBC encryption

(d) KL graph partition

Figure 6.4: Tagged information spread throughout execution

information throughout executions of four programs that make extensive use of pointer manipulations with our tagged pointer tracking turned on. These programs are bzip2, gzip, AES CBC encryption, and the Kernighan-Lin (KL) graph partitioning tool, obtained from the pointer-intensive benchmark suite [6]. The bzip2 and gzip programs make extensive use of pointers in their compression algorithms. Part of the AES encryption algorithm requires lookups into a static table, known as the S-box. For these three programs, tagged pointer tracking is required to accurately track information flow through the program, or else this tagged information is lost due to the indirect

memory accesses that occur through pointer and array arithmetic and table lookups. In addition, the KL algorithm implementation utilizes data structures like arrays and linked lists extensively for graph representation.

The measurements of information spread for pointer-intensive workloads can be seen in Figure 6.4 for x86, x86_64, and ARM. Figures 6.4(a), 6.4(b), and 6.4(c) show similar results for each ISA in terms of the number of tagged bytes in memory, but we can see offsets in instruction counts that highlights one of the differences of these CISC vs. RISC implementations.

Figures 6.4(a) and 6.4(b) show the results of compressing the same file, which was approximately 300 bytes. These figures show the extent of information spread for these workloads; the peak number of tagged bytes reaches 14x the original file size for bzip2 on average across each ISA, and 6x the original file size for gzip on average across each ISA. For bzip2, the drastic growth in tagged data occurs as soon as the block sorting starts in the algorithm. For gzip, there is a more gradual increase in tagged data as soon as the compression function begins compressing the data. These patterns are indicative of the complex manipulations that are made on files as the tagged data flows through these compression algorithms. On the contrary, while many complex manipulations occur on files through AES encryption, Figure 6.4(c) shows that the amount of tagged data increases just over 2x for each ISA. Overall, these three pointer-intensive algorithms show similar patterns of state explosion for information flow tracking, regardless of the underlying ISAs that we've evaluated.

Figure 6.4(d) on the other hand shows major discrepancies in the amount of tagged information across the various ISAs. For this experiment, we processed a file of size 1260 bytes. For x86 and ARM, we can see an initial increase of tagged information followed by a gradual increase up to a maximum of 1.5x and 4.1x the original tagged

data, respectively. For x86_64, it is clear that a form of state explosion occurs causing the amount of tagged information to spread dramatically, reaching 11x the amount of original tagged data. Looking more closely at the x86_64 instance, we found that this initial explosion occurs inside of C library functions. One reason for this state explosion is that tagged data propagated to a global variable or base pointer, resulting in subsequent propagation with every access of that variable or base pointer. The fact that this explosion occurs inside of the C library implementation explains why we see the discrepancies across ISAs.

Chapter 7

Discussion

Currently, our system provides the capability to perform dynamic information flow tracking for several of the major ISAs supported by the QEMU binary translator: x86, x86_64, and ARM. It is trivial to support more of these ISAs since we already have the ability to translate from the TCG IR to the LLVM IR. We plan to support more of the ISAs included in QEMU as future work.

One limitation of the QEMU user mode emulator is that there is limited support for multi-threaded programs. To deal with this, we plan to extend our system to support the QEMU whole-system emulator. We also plan to extend our coupled execution and analysis approach to work with systems at runtime.

In addition to supporting the QEMU whole-system emulator, we also plan to fully support information flow tracking for all instructions that QEMU implements for each supported ISA. This is challenging because for each ISA that QEMU supports, there are several hundred instructions that are carried out in QEMU's C code helper functions. However, our combined static and dynamic analysis of helper functions that we've described in Section 5.5 provides the basis for this solution.

Chapter 8

Conclusion and Future Work

In this thesis, we have presented PIRATE, an architecture-independent information flow tracking framework that enables dynamic information flow tracking at the binary level for several different ISAs. In addition, our combined static and dynamic analysis of helper function C code enables us to track information that flows through these complex instructions for each ISA. PIRATE enables us to decouple all of the useful applications of dynamic information flow tracking from specific ISAs without requiring source code of the programs we are interested in analyzing. To demonstrate the utility of our system, we have applied it in three security-related contexts; enforcing information flow policies, characterizing algorithms, and diagnosing sources of state explosion. Using PIRATE, we can continue to build on the usefulness of dynamic information flow tracking by bringing these security applications to a multitude of ISAs without requiring extensive domain knowledge of each ISA, and without the extensive implementation time required to support each ISA.

8.1 Summary of Contributions

PIRATE advances the state of the art in dynamic information flow tracking by providing an approach that works on binaries of several different ISAs without requiring source code. The contributions we have made with this thesis are the following:

- We have introduced a framework that leverages dynamic binary translation to enable architecture-independent dynamic analyses in terms of the LLVM intermediate representation. We have also defined a language for precisely expressing information flow of this IR at the byte level. Our language combined with the use of label sets allows us to gather detailed information about how programs operate and process data.

- We have implemented a combined static/dynamic analysis to be applied to the C code of the binary translator for complex ISA-specific instructions that do not fit within the IR, enabling the automated analysis and inclusion of these instructions in our framework. This takes a significant burden off of developers of information flow tracking systems since this analysis is automated, saving software analysts the effort of carefully modeling hundreds of instructions for several different ISAs.

- We have evaluated of our information flow tracking framework for x86, x86_64, and ARM, highlighting three security-related applications: 1) enforcing information flow policies, 2) characterizing algorithms by information flow, and 3) diagnosing sources of state explosion for each ISA. These evaluations provide the basis for more advanced applications of dynamic information flow tracking on other ISAs besides the popular x86.

8.2 Future Work

PIRATE is currently decoupled between execution and analysis. While this approach provides flexibility in performing different types of instrumentation on a single execution, we plan to extend this system to work at run time as well. This will enable live analyses which can be more useful from a security perspective, especially in the context of detection and prevention of exploits. As we adapt PIRATE to work at run time, we will also need to focus on further optimizations of information flow operations so it can work as efficiently as possible.

Our combined static/dynamic analysis of a subset of QEMU helper functions provides the basis for automatically supporting all helper functions for each QEMU ISA. Our goal is to extend this analysis to support complex instructions for each ISA including privileged instructions, floating point instructions, SIMD instructions, and more. Extending the completeness of our system in this way will enable more detailed security analyses of programs and operating systems for all of the ISAs that QEMU supports.

With these enhancements, we will have the ability to perform detailed security analyses for entire operating systems, regardless of the ISA that they are compiled to run on. For instance, combining PIRATE with the Android emulator (which is based on QEMU) could provide a platform for taint-based analyses, and other IR-based analyses to assist analysts with the ever-increasing problem of Android malware. Additionally, we also have the opportunity to perform taint-based vulnerability discovery for embedded architectures similar to the approaches taken with BuzzFuzz [19] and TaintScope [40]. Potential systems to be analyzed include networked systems that run on QEMU (such as the OpenWRT router on MIPS), and other embedded operating systems (such as VxWorks or embedded Linux distributions). Our

architecture-independent approach will allow us to perform important analyses for these embedded systems ISAs, where support for dynamic information flow tracking with existing systems is limited.

Bibliography

[1] Clang: A c language family frontend for llvm. http://clang.llvm.org.

[2] Mcafee threats report: Third quarter 2012. http://www.mcafee.com/us/resources/reports/rp-quarterly-threat-q3-2012.pdf.

[3] National vulnerability database. http://nvd.nist.gov.

[4] Openssl cryptography and ssl/tls toolkit. http://openssl.org.

[5] Erik Altman, Michael Gschwind, Sumedh Sathaye, Stephen Kosonocky, Arthur Bright, Jason Fritts, Paul Ledak, D Appenzeller, Craig Agricola, and Z Filan. BOA: The Architecture of a Binary Translation Processor. Technical Report RC21665, IBM, 1999.

[6] Todd Austin, Scott E. Breach, and Gurindar S. Sohi. Efficient detection of all pointer and array access errors. Technical report, University of Wisconsin-Madison, December 1993.

[7] Fabrice Bellard. QEMU, a Fast and Portable Dynamic Translator. In *USENIX Annual Technical Conference*, 2005.

[8] Derek Bruening, Timothy Garnett, and Saman Amarasinghe. An infrastructure for adaptive dynamic optimization. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, CGO '03, pages 265–275, Washington, DC, USA, 2003. IEEE Computer Society.

[9] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J Schwartz. BAP: A Binary Analysis Platform. In *Computer Aided Verification*, July 2011.

[10] Juan Caballero, Pongsin Poosankam, Christian Kreibich, and Dawn Song. Dispatcher: enabling active botnet infiltration using automatic protocol reverse-engineering. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, 2009.

[11] Shuo Chen, Jun Xu, Nithin Nakka, Zbigniew Kalbarczyk, and Ravishankar K Iyer. Defeating Memory Corruption Attacks via Pointer Taintedness Detection. In *International Conference on Dependable Systems and Networks*, 2005.

[12] Anton Chernoff, Mark Herdeg, Ray Hookway, Chris Reeve, Norman Rubin, Tony Tye, S Bharadwaj Yadavalli, and John Yates. FX!32: A Profile-directed Binary Translator. *IEEE Micro*, 18(2):56–64, 1998.

[13] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. S2E: a platform for in-vivo multi-path analysis of software systems. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2011.

[14] Jim Chow, Ben Pfaff, Tal Garfinkel, Kevin Christopher, and Mendel Rosenblum. Understanding data lifetime via whole system simulation. In *Proceedings of the 13th USENIX Security Symposium*, 2004.

[15] James Clause, Wanchun Li, and Alessandro Orso. Dytan: a generic dynamic taint analysis framework. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, 2007.

[16] James C Dehnert, Brian K Grant, John P Banning, Richard Johnson, Thomas Kistler, Alexander Klaiber, and Jim Mattson. The Transmeta Code Morphing Software: using speculation, recovery, and adaptive retranslation to address real-life challenges. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, CGO '03, pages 15–24, Washington, DC, USA, 2003. IEEE Computer Society.

[17] Kemal Ebcioglu and Erik R Altman. DAISY: Dynamic Compilation for 100% Architectural Compatibility. Technical Report RC20538, IBM, 1996.

[18] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N Sheth. TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, 2010.

[19] Vijay Ganesh, Tim Leek, and Martin Rinard. Taint-based directed whitebox fuzzing. In *IEEE 31st International Conference on Software Engineering*, 2009.

[20] Min Gyung Kang, Stephen McCamant, Pongsin Poosankam, and Dawn Song. DTA++: Dynamic Taint Analysis with Targeted Control-Flow Propagation. In

Proceedings of the 18th Annual Network and Distributed System Security Symposium, 2011.

[21] Vasileios P Kemerlis, Georgios Portokalidis, Kangkook Jee, and Angelos D Keromytis. libdft: Practical Dynamic Data Flow Tracking for Commodity Systems. In *Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments*, 2012.

[22] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization*, 2004.

[23] Tim Leek, Graham Baker, Ruben Brown, Michael Zhivich, and Richard Lippman. Coverage Maximization Using Dynamic Taint Tracing. Technical Report 1112, MIT Lincoln Laboratory, March 2007.

[24] 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. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2005.

[25] Noe Lutz. *Towards Revealing Attackers Intent by Automatically Decrypting Network Traffic*. Master's thesis, ETH Zurich, 2008.

[26] Shashidhar Mysore, Bita Mazloom, Banit Agrawal, and Timothy Sherwood. Understanding and visualizing full systems with data flow tomography. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2008.

[27] George C Necula, Scott Mcpeak, Shree P Rahul, and Westley Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *International Conference on Compiler Construction*, pages 213–228, 2002.

[28] Nicholas Nethercote and Julian Seward. How to shadow every byte of memory used by a program. In *Proceedings of the 3rd international conference on Virtual execution environments*, VEE '07, pages 65–74, New York, NY, USA, 2007. ACM.

[29] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2007.

[30] Georgios Portokalidis, Asia Slowinska, and Herbert Bos. Argos: an emulator for fingerprinting zero-day attacks. In *Proceedings of ACM SIGOPS EUROSYS 2006*, April 2006.

[31] Thomas Reps, Gogul Balakrishnan, and Junghee Lim. Intermediate-representation recovery from low-level code. In *Proceedings of the 2006 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation*, 2006.

[32] Prateek Saxena, R Sekar, and Varun Puranik. Efficient fine-grained binary instrumentation with applications to taint-tracking. In *Proceedings of the 6th IEEE/ACM International Symposium on Code Generation and Optimization*, 2008.

[33] Asia Slowinska and Herbert Bos. Pointless Tainting? Evaluating the Practicality of Pointer Tainting. In *EuroSys*, April 2009.

[34] Dawn Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Prateek Saxena. BitBlaze: A new approach to computer security via binary analysis. In *Proceedings of the 4th International Conference on Information Systems Security*, December 2008.

[35] Amitabh Srivastava and Alan Eustace. ATOM: a system for building customized program analysis tools. In *Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, PLDI '94, pages 196–205, New York, NY, USA, 1994. ACM.

[36] William Stallings and Lawrie Brown. *Computer Security: Principles and Practice*. Pearson Prentice Hall, 2008.

[37] G Edward Suh, Jae W Lee, David Zhang, and Srinivas Devadas. Secure program execution via dynamic information flow tracking. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2004.

[38] Omer Tripp, Marco Pistoia, Stephen J. Fink, Manu Sridharan, and Omri Weisman. Taj: effective taint analysis of web applications. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2009.

[39] Guru Venkataramani, Ioannis Doudalis, Yan Solihin, and Milos Prvulovic. FlexiTaint: A programmable accelerator for dynamic taint propagation. In *Proceedings of the 14th International Symposium on High Performance Computer Architecture*, 2008.

[40] Tielei Wang, Tao Wei, Guofei Gu, and Wei Zou. TaintScope: A Checksum-Aware Directed Fuzzing Tool for Automatic Software Vulnerability Detection. In *Proceedings of the 31st IEEE Symposium on Security & Privacy*, May 2010.

[41] Zhi Wang, Xuxian Jiang, Weidong Cui, Xinyuan Wang, and Mike Grace. Reformat: automatic reverse engineering of encrypted messages. In *Proceedings of the 14th European Conference on Research in Computer Security*, 2009.

[42] Wei Xu, Sandeep Bhatkar, and R Sekar. Taint-enhanced policy enforcement: a practical approach to defeat a wide range of attacks. In *Proceedings of the 15th USENIX Security Symposium*, 2006.

[43] Qin Zhao, Derek Bruening, and Saman Amarasinghe. Umbra: efficient and scalable memory shadowing. In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, CGO '10, pages 22–31, New York, NY, USA, 2010. ACM.