The Pennsylvania State University

The Graduate School

Information Sciences and Technology

PIPELINED SYMBOLIC TAINT ANALYSIS

A Dissertation in

College of Information Sciences and Technology

by

Jiang Ming

Submitted in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

August 2016

The dissertation of Jiang Ming was reviewed and approved* by the following:

Dinghao Wu

Assistant Professor of Information Sciences and Technology

Dissertation Advisor, Chair of Committee

Peng Liu

Professor of Information Sciences and Technology

Committee Member

Sencun Zhu

Associate Professor of Computer Science and Engineering

Committee Member

Patrick Drew McDaniel

Professor of Computer Science and Engineering

Outside Member

Andrea Tapia

Associate Professor of Information Sciences and Technology

Director of Graduate Programs

*Signatures are on file in the Graduate School.

Abstract

As people rely more and more on computers, building a secure computing environment becomes an important research topic. However, many computer programs exist security bugs, and advanced attacking techniques have been discovered to break into a computer. Therefore, to protect computer systems, a large number of security-relevant software analysis approaches have been developed to detect the intrusions and analyze the vulnerabilities and malware. One class of security analysis, *taint analysis*, is to trace information flow along program execution path. The basic idea of taint analysis is to insert some kind of tags or labels for user input and propagate the tags along a program execution path, and then check the taint status at certain critical locations. Taint analysis can be seen as a form of information flow analysis since we can observe how the data information is being copied and modified.

The multifaceted benefits of taint analysis have led to its wide adoption in security applications, such as software attack detection, attack provenance investigation, and data lifetime analysis. Static taint analysis propagates taint values following all possible paths with no need for concrete execution, but is generally less accurate than dynamic analysis. Unfortunately, the high runtime overhead imposed by dynamic taint analysis makes it impractical in many scenarios. The key obstacle is the strict coupling of program execution and taint tracking logic code. To alleviate this performance bottleneck, recent work seeks to offload taint analysis from program execution and run it on a different CPU offline or a spare core in parallel. However, since the taint analysis has heavy data and control dependencies on the program execution, the massive data in recording or transformation overshadows the benefit of decoupling. In this thesis, we continue this line of research to take advantage of ubiquitous multi-core platforms.

First, we propose *StraightTaint*, a hybrid taint analysis tool that completely decouples the program execution and taint analysis. StraightTaint allows very lightweight logging, resulting in much lower execution slowdown, while still permitting us to perform full-featured offline taint analysis, including bit-level and multi-tag taint analysis. StraightTaint relies on very lightweight logging of the execution information

to reconstruct a straight-line code, enabling an offline symbolic taint analysis without frequent data communication with the application. While StraightTaint does not log complete runtime or input values, it is able to precisely identify the causal relationships between sources and sinks, for example. Our experimental results show that on average StraightTaint can speed up application execution performance 3.32 times on a set of utility programs and 3.25 times on SPEC2006.

The collected data during online logging for some long running program could be still too large to be saved up for offline analysis. To address this limitation, we propose *TaintPipe*, a novel technique for parallelizing and pipelining taint analysis. TaintPipe produces compact control flow profiles and spawns multiple threads as different stages of a pipeline to carry out symbolic taint analysis in parallel. Our experiments show that TaintPipe imposes low overhead on application runtime performance and accelerates taint analysis significantly. Compared to a state-of-the-art inlined dynamic data flow tracking tool, TaintPipe achieves 2.38 times speedup for taint analysis on SPEC2006 and 2.43 times for a set of common utilities, respectively. In addition, we demonstrate the strength of TaintPipe with a set of security applications.

Last but not least, due to the design, both StraightTaint and TaintPipe are an ideal fit for *ex post facto* security applications, such as computer forensic analysis and reverse engineering. To make the techniques described in this thesis applicable to a broader range of such applications, we have integrated the techniques proposed in this thesis into the tasks of binary code control flow profiling, encoding function detection, and speeding up semantics-based binary diffing.

Contents

List of Figures

List of Tables

Acknowledgments

Firstly, I would like to express my sincere gratitude to my advisor Dr. Dinghao Wu, who has been a great mentor for me. I would like to thank him for encouraging my research and for allowing me to grow as an independent researcher. His advice on both research as well as on my career has been priceless. I could not have imagined to have a better advisor and mentor for my Ph.D. study. I would also like to thank the rest of my thesis committee: Dr. Peng Liu, Dr. Sencun Zhu, and Dr. Patrick D. McDaniel, for their insightful comments and encouragement, but also for the hard questions which helped me to broaden my research from various perspectives.

A special thanks to my family. Words cannot express how grateful I am to my mother and father for all of the sacrifices that they have made on my behalf. I would also like to thank my lab fellows for the stimulating discussions, for the sleepless nights we worked together before deadlines, and for all the fun we had in the last five years. In the end, I would like express appreciation to my beloved fiancee, Miss Chang Liu. Thanks for supporting me for everything, and especially I cannot thank her enough for encouraging me throughout this experience.

This dissertation is based upon research supported in part by the U.S. National Science Foundation under award numbers CNS-1223710 and CCF-1320605, and the U.S. Office of Naval Research under award numbers N00014-13-1-0175 and N00014-16-1-2265.

Chapter 1
Introduction

As people rely more and more on computers, building a secure computing environment becomes an important research topic. However, many computer programs exist security vulnerabilities, and advanced attacking techniques have been discovered to break into a computer. For example, Ubuntu Linux has over 99,000 known bugs [1], and many of them are really serious security vulnerabilities. Software vulnerabilities may allow an attacker to inject attack code and cause the compromised machine to run the malicious code. Another fact is the number of malicious software is rising significantly in recent years. By the end of 2015, nearly half a billion malicious software are in circulation [2]. Therefore, to protect computer systems, a large number of security-relevant software analysis approaches have been developed to detect the intrusions and analyze the vulnerabilities and malware.

Many security-relevant software analysis tools work on binary code instead of source code. Binary code analysis is very attractive from the security viewpoint. First, In many cases, we do not have access to the source code of the program we care about, such as commercial off-the-shelf software and malicious software. The binary code becomes the only available resource to be analyzed. Second, Binary analysis provides the ground truth about program behavior because computers execute binaries, not source code. One

class of security analysis is to monitor an application execution and perform runtime security analysis. Compared to the pre-execution tools, runtime security analysis is more accurate since it only considers the real path taken at the run time. It has the advantage of observing the actual dynamic state (such as program inputs and memory values). With these concrete execution state information, runtime security analysis has the better resilience to obfuscation methods as well. Unfortunately, many of the most useful runtime security analysis tools are expensive. They may cause an application to execute too slowly.

As a result, security researchers who wish to detect the intrusions and analyze the vulnerabilities are faced with a dilemma: powerful runtime security analysis may introduce very high runtime overhead, but other weaker security analysis may not detect the intrusions or vulnerabilities. Therefore, building both efficient and accurate runtime security analysis becomes extremely important. This thesis aims to address the performance issue of dynamic taint analysis, which is a very useful runtime security analysis to trace the information flow along program execution path. Dynamic taint analysis has been widely applied in software attack detection, information flow control , data leak detection and malware analysis. However, dynamic taint analysis also suffers from high performance penalty. The slowdown introduced by conventional dynamic taint analysis tools can easily go up to 30X times. The high overhead has limited its adoption in production systems.

In this thesis, I will present my work to improve the performance of dynamic taint analysis by utilizing the pervasive multi-core architectures. Our techniques decouple the expensive taint analysis from application to either offline execution or multiple pipeline stages running in parallel. With the developed technique, we can enable broader adoptions of information flow tracking technology in practice. Next, I will first introduce

the background of taint analysis, the performance bottleneck of dynamic taint analysis, and the challenges of dynamic taint analysis decoupling.

1.1 Taint Analysis

The idea of taint analysis is very similar to biology cell dyeing [3]. Cell dyeing is frequently used in biology to color some materials, and then trace their movement or highlight some structures. For example, We can color a parent cell to observe how the parent cell divides into two daughter cells. The basic idea of taint analysis is to insert some kind of tags or labels for user input and propagate the tags along a program execution path, and then check the taint status at certain critical locations. Taint analysis can be seen as a form of information flow analysis. We can observe how the data information is being copied and modified. Taint Analysis has three main parts: taint seed, taint propagation policy, and taint sink. Taint seed defines which data to tag, and how to tag it. Typically, we label the input from untrusted sources as tainted, such as different types of documents, network protocols, keyboard inputs, and system call return values. We can label a single tag to the whole input or assign different taint tags to represent the different structures of the input. A taint propagation policy decides how to propagate taint tags. A policy can consider both data and control flow dependencies or only data flow dependencies. A taint sink indicates where and how tags should be checked. Taint analysis has been shown to be effective in dealing with a wide range of security problems, including software attack prevention [4, 5], information flow control [6, 7], data leak detection [8], and malware analysis [9, 10], to name a few.

We can perform taint analysis on C, Java, Python, and other programming languages. However, taint analysis on binary code is very attractive from a security viewpoint because we do not require the source code of the application. We can even trace The

information flow in the malicious binary code. On the other side, due to the lack of higher-level semantics, such as data types or data structures, it is also very challenging to develop taint analysis on binary code. We have to track taint tags for assembly instructions. And additional shadow memory is required to store the taint tags being tracked. The minimum amount of memory required is one bit for every tagged piece of data.

Generally, taint analysis on binary code comes two flavors. Static taint analysis (STA) [11–13] is performed prior to execution and therefore it has no impact on runtime performance. STA has the advantage of considering multiple execution paths, but at the cost of potential imprecision. For example, STA may result in either under-tainting or over-tainting [14] when merging results at control flow confluence points. Dynamic taint analysis (DTA) [4, 15, 16], in contrast, propagates taint as a program executes, which is more accurate than static taint analysis since it only considers the actual path taken at run time. To directly work with program binaries, dynamic taint analysis is typically developed based on Pin [17], Qemu [18] or other dynamic binary instrumentation (DBI) platforms to insert taint tracking code just in time. Typically, we insert taint tracking code into binary code right before it runs (as shown in Figure 1.1). In this way, we can detect the violation of the security policy before it really happens. Following this style, the taint tracking code runs as part of the normal instruction stream after being injected. However, the high runtime overhead imposed by dynamic taint propagation has severely limited its adoption in production systems. The slowdown incurred by conventional dynamic taint analysis tools [4, 15] can easily go beyond 30X times. Even with the state-of-the-art DTA tool based on Pin [19], typically it still introduces more than 6X slowdown.

The key obstacle to further improving the performance of dynamic taint analysis is the tight coupling of program execution and taint tracking logic code [20]. Taint analysis has to maintain a shadow memory to map instruction operands to their corresponding

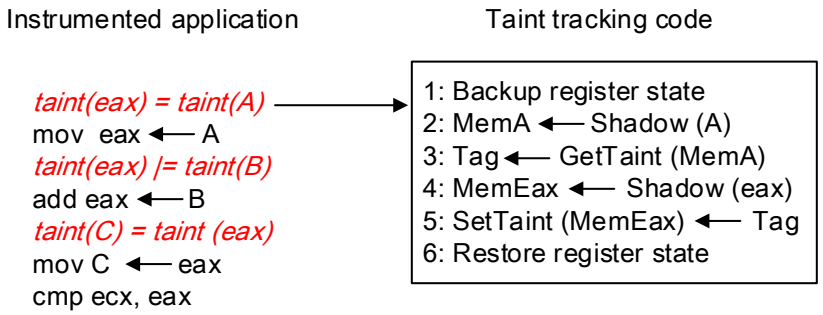

Figure 1.1. Dynamic taint analysis inserts taint tracking code just in time.

taint tags. To propagate one taint tag between different residences, it typically takes 6–8 additional instructions [21]. Especially when dynamic taint analysis runs on the same CPU core, application execution and taint analysis have to compete for CPU cycles, registers and cache space. For example, if the taint tracking code modifies a general purpose register, this register must be saved before the execution of the taint tracking code and then restore the saved value later. As a result, dynamic taint analysis may slow down performance by 10X $\sim$ 30X! The high performance penalty comes from the strict coupling of program execution and taint tracking code; that is, the runtime execution has to frequently switch between normal application execution and taint tracking code.

1.2 Challenges

A nature idea to reduce the overhead is to decouple the expensive taint analysis from application execution. The proliferation of multi-core systems has inspired researchers to decouple taint tracking logic onto spare cores in order to improve performance. One direction is hardware-assisted approaches. For example, Speck [22] needs OS level support for speculative execution and rollback. Ruwase et al. [23] employ a customized hardware for logging a program trace and delivering it to other idle cores for inspection. Nagarajan et al. [24] utilize a hardware first-in first-out buffer to speed up communication

between cores. Although they can achieve an appealing performance, the requirement of special hardware prevents them from being adopted using commodity hardware. Another direction is software-only methods that work with binary executables on commodity multi-core hardware. These software-only solutions typically rely on dynamic binary instrumentation (DBI) to decouple dynamic taint analysis from program execution. The existing software-only work can be roughly classified into two categories. The first category relies on the pervasive multi-core systems to parallelize dynamic taint analysis by logging runtime values that are needed for taint analysis in another core [25, 26].

However, since taint analysis has strong serial data and control dependencies on the program execution, the parallelized taint analysis need to be frequently synchronized for data communication (e.g., control flow directions and memory addresses). The second category first records the application execution and then replay the taint analysis on a different CPU [27–29]. Similar to the limitation of the first category, the large online logging data is a barrier to achieving the expected performance gains. Furthermore, the taint tracking execution is typically slower than the application execution. Since we need extra instructions to access shadow memory and propagate taint tags, the delay for each taint operation could be accumulated, leading to a very large delay to the application execution. On the other side, application execution has to wait for more and more time when doing synchronization with the decoupled taint analysis.

Recent work ShadowReplica [26] creates a secondary shadow thread from primary application thread to run DTA in parallel. ShadowReplica conducts an offline optimization to generate optimized DTA logic code, which reduces the amount of information that needs to be communicated, and thus dramatically improves the performance. However, as we will show later, the performance improvement achieved by this "primary & secondary" thread model is fixed and cannot be improved further

when more cores are available. Furthermore, in many security related tasks (e.g., binary de-obfuscation and malware analysis), precise static analysis for the offline optimization needed by ShadowReplica may not be feasible.

1.3 Thesis Organization

The rest of the thesis is organized as follows. Chapter 2 presents an overview about our proposed research work. Chapter 3 provides the extensive related work on taint analysis. Chapter 4 presents the design and evaluation of our decoupled offline symbolic taint analysis. Chapter 5 discusses our pipelined symbolic taint analysis in detail. Chapter 6 demonstrates three appealing applications of our proposed techniques. Limitations and future work are introduced in Chapter 7. We conclude the thesis in Chapter 8.

Chapter 2
Overview

This thesis presents techniques that can decouple and parallelize an important security analysis approach to reduce the high runtime overhead. The topic of my research work is pipelined symbolic taint analysis. In Chapter 1, I have introduced the taint analysis background, the advantage of dynamic taint analysis and its performance bottleneck, and the challenges of taint parallelization. To address these issues, we have developed techniques to completely decouple the expensive taint analysis to to the offline analysis and parallelize taint analysis in pipeline style. We have applied our techniques to speed up multiple tedious reverse engineering tasks. Next I will start with a general overview about our proposed research work.

2.1 Decoupled Offline Symbolic Taint Analysis

To lower the high performance overhead, multiple methods have been proposed to offload taint tracking code to a separate core or a different CPU. The existing work can be roughly classified into two categories. The first category relies on the pervasive multi-core systems to parallelize dynamic taint analysis by logging runtime values that are needed for taint analysis in another core [22, 23, 26, 30]. However, since taint analysis has

strong serial data and control dependencies on the program execution, the parallelized taint analysis need to be frequently synchronized for data communication (e.g., control flow directions and memory addresses), either through customized hardware [22, 23] or shared memory [26, 30]. The second category first records the application execution and then replay the taint analysis on a different CPU [27–29, 31]. Similar to the limitation of the first category, the large online logging data is also a barrier to achieving the expected performance gains.

In this thesis, we first propose *StraightTaint*, a hybrid static and dynamic method that achieves very lightweight logging, resulting in much lower execution slowdown, while still permitting us to perform complete offline taint analysis with incomplete inputs. In principle, StraightTaint belongs to the aforementioned second category of decoupling DTA approaches. Therefore, StraightTaint is an ideal fit for *ex post facto* security applications. In StraightTaint, we do not log all runtime values. Instead, we record control flow profiling and execution state when taint seeds are first introduced, which can be very lightweight. Based on the logged branching information, we construct a straight-line code trace for the offline taint analysis. The taint seeds are marked as symbolic variables, and the taint propagation is like symbolic execution on the constructed straight-line code. With the initial execution state and the straight-line code, most addresses of memory load and store operations are computable. Symbolic memory indices can be restricted to a small range by solving the path conditions. Compared to a pure static approach, StraightTaint can still deliver a similar level of precision as dynamic taint analysis. For example, we are able to correctly identify the complicated causal relationships among multiple sources and sinks (see Section 4.6), while static taint analysis fails in such cases.

Restricted by the computing resources, conventional DTA exhibits several drawbacks in terms of incomplete taint propagation strategies. First, since multi-tag taint propagation

consumes more shadow memory and introduces much higher runtime overhead, most DTA tools choose single-tag propagation as default [4, 8, 16, 19, 21, 23]. However, multi-tag taint analysis is indispensable to many reverse engineering tasks, such as recovering the structure of an unknown protocol format [32] and detecting encoding functions in malware by counting different tainted input bytes [9]. Second, when handling the complicated x86 arithmetic and logic operation instructions (e.g., `add` and `xor`), previous DTA tools typically adopt some simple but conservative propagation strategies for better performance. One example is the prevalent "short circuiting" method: the destination operand is tainted if any of the source operands is tainted. As we will show, these conservative solutions will result in precision loss in many scenarios. As StraightTaint has completely offloaded the taint logic code to the offline analysis, another benefit becomes visible: StraightTaint's offline taint analysis is flexible to support *full-featured* taint propagation strategies. For example, supporting bit-level [31] or multi-tag taint analysis is straightforward in our approach. Each symbolic bit or variable can naturally represent a taint tag with negligible additional overhead. Also, our symbolic execution style taint propagation can faithfully simulate the specific semantics of an instruction. Furthermore, based on symbolic taint analysis on the straight-line code, we introduce a new concept, *Conditional Tainting*; that is, StraightTaint is able to identify precisely the causal data flow relations between sources and sinks, under what *conditions*. In this way, new inputs and runtime values can be mapped to the existing analysis results in certain scenarios so that the new analysis can be more proactive.

We have developed a prototype of StraightTaint, a hybrid taint analysis approach that completely decouples the program execution and taint analysis. Our implementation is based on Pin [17], for the effective parallelization of runtime logging, and BAP [33], for precise offline symbolic taint analysis with incomplete inputs. We have performed

comparative studies on a number of applications such as common utility programs, SPEC2006, and real-life software vulnerabilities. The results show that StraightTaint can achieve a similar level of precision as dynamic taint analysis, but with much lower online execution slowdown. The performance experiments show that StraightTaint imposes a small overhead on application execution performance, with up to 3.25 times improvements to SPEC2006 on average. Offline taint analysis takes approximately the same amount of time as an advanced DTA tool. We also demonstrate StraightTaint's value in supporting multi-tag taint propagation and conditional tainting in an attack provenance investigation task. Such experimental evidence shows that StraightTaint can be applied to various large-scale *ex post facto* security applications.

2.2 Pipelined Symbolic Taint Analysis

The collected data generated by StraightTaint's online logging for some long running program could be still too large to be saved up for offline analysis. A possible solution is to decouple taint analysis via parallelization, which performs taint analysis on the fly without the log storage. Recent work ShadowReplica [26] creates a secondary shadow thread from primary application thread to run DTA in parallel. ShadowReplica conducts an offline optimization to generate optimized DTA logic code, which reduces the amount of information that needs to be communicated, and thus dramatically improves the performance. However, as we will show later, the performance improvement achieved by this "primary & secondary" thread model is fixed and cannot be improved further when more cores are available. Furthermore, in many security related tasks (e.g., binary de-obfuscation and malware analysis), precise static analysis for the offline optimization needed by ShadowReplica may not be feasible.

In the thesis, we exploit another style of parallelism, namely pipelining, to perform

taint analysis on-the-fly without the log storage. We propose a novel technique, called TaintPipe, for parallel data flow tracking using *pipelined symbolic taint analysis*. In principle, TaintPipe falls within the second category of taint decoupling work classified above. Essentially, in TaintPipe, threads form multiple pipeline stages, working in parallel. The execution thread of an instrumented application acts as the source of pipeline, which records information needed for taint pipelining, including the control flow data and the concrete execution states when the taint seeds are first introduced. To further reduce the online logging overhead, we adopt a compact profile format and an N-way buffering thread pool. The application thread continues executing and filling in free buffers, while multiple worker threads consume full buffers asynchronously. When each logged data buffer becomes full, an inlined call-back function will be invoked to initialize a taint analysis engine, which conducts taint analysis on a segment of straight-line code concurrently with other worker threads. Symbolic memory access addresses are determined by resolving indirect control transfer targets and approximating the ranges of the symbolic memory indices.

To overcome the challenge of propagating taint tags in a segment without knowing the incoming taint state, TaintPipe performs *segmented symbolic taint analysis*. That is, the taint analysis engine assigned to each segment calculates taint states symbolically. When a concrete taint state arrives, TaintPipe then updates the related taint states by replacing the relevant symbolic taint tags with their correct values. We call this *symbolic taint state resolution*. According to the segment order, TaintPipe sequentially computes the final taint state for every segment, communicates to the next segment, and performs the actual taint checks. Optimizations such as function summary and taint basic block cache offer enhanced performance improvements. Moreover, different from previous DTA tools, supporting bit-level and multi-tag taint analysis are straightforward for TaintPipe.

TaintPipe does not require redesign of the structure of shadow memory; instead, each taint tag can be naturally represented as a symbolic variable and propagated with negligible additional overhead.

We have developed a prototype of TaintPipe, a pipelined taint analysis tool that decouples program execution and taint logic, and parallelizes taint analysis on straight-line code segments. Our implementation is built on top of Pin [17], for the pipelining framework, and BAP [33], for symbolic taint analysis. We have evaluated TaintPipe with a variety of applications such as the SPEC CINT2006 benchmarks, a set of common utilities, a list of recent real-life software vulnerabilities, malware, and cryptography functions. The experiments show that TaintPipe imposes low overhead on application runtime performance. Compared with a state-of-the-art inlined dynamic taint analysis tool, TaintPipe achieves overall 2.38 times speedup on SPEC CINT2006, and 2.43 times on a set of common utility programs, respectively. The efficacy experiments indicate that TaintPipe is effective in detecting a wide range of real-life software vulnerabilities, analyzing malicious programs, and speeding up cryptography function detection with multi-tag propagation. Such experimental evidence demonstrates that TaintPipe has potential to be employed by various applications in production systems.

2.3 Applications to Reverse Engineering

A byproduct of the StraightTaint and TaintPipe projects is that we design a novel binary code control flow profiling tool, *BinCFP*, which allows efficient and complete binary code control flow profiling on ubiquitous multi-core platforms. In many tasks of reverse engineering and binary code analysis (e.g., hybrid disassembly, resolving indirect jump, and decoupled taint analysis), the knowledge of detailed dynamic control flow can be of great value. However, the high runtime overhead beset the complete collection of dynamic

control flow. The previous efforts on efficient path profiling cannot be directly applied to the obfuscated binary code in which an accurate control flow graph is typically absent. To address these challenges, we present *BinCFP*, an efficient multi-threaded binary code control flow profiling tool by taking advantage of pervasive multi-core platforms. BinCFP relies on dynamic binary instrumentation to work with the unmodified binary code. The key of BinCFP is a multi-threaded fast buffering scheme that supports processing trace buffers asynchronously. To achieve better performance gains, we also apply a set of optimizations to reduce control flow profile size and instrumentation overhead. Our design enables the complete dynamic control flow collection for an obfuscated binary execution. We have implemented BinCFP on top of Pin. The comparative experiments on SPEC2006 and obfuscated common utility programs show BinCFP outperforms the previous work in several ways. In addition, BinCFP's control flow profile sizes are only about 49.2% that of the conventional design.

In addition to runtime policy enforcement [4, 16], taint analysis on binary code is also broadly used in *ex post facto* security applications, such as attack provenance investigation [34, 35], computer forensic analysis [36], malware analysis [9, 10], and reverse engineering [32, 37, 38]. Due to the decoupled analysis style, both StraightTaint and TaintPipe are an ideal fit for such post fact security analysis tasks. To make the techniques described in this thesis applicable to a broader range of such applications, we have integrated the decoupled symbolic multi-tag taint analysis into the following two tasks.

1. Identify possible cryptography functions in the binary code by observing the input-output dependencies with multi-tag taint analysis. Our proposed decoupled symbolic taint analysis is naturally a multi-tag taint analysis method. We demonstrate this capability by detecting cryptography functions in binary with

little additional overhead.

2. Utilize multi-tag taint analysis to speed up semantics-based binary difference analysis. Semantics-based binary diffing suffers from significant overhead, hindering it from analyzing large numbers of malware samples. Our efficient decoupled symbolic taint analysis can greatly reduce the number of possible basic block matches.

2.4 Contributions

The contributions of this thesis are summarized as follows.

1. We propose StraightTaint, with a very lightweight logging method to construct straight-line code and thus completely decouple dynamic taint analysis for offline symbolic taint analysis. StraightTaint greatly reduces the program execution slowdown yet can compete with dynamic taint analysis with a similar level of precision.

2. The limitation of previous decoupling taint work is inefficiently collecting and transferring data from the executing application to the analysis module. We demonstrate that StraightTaint offline analysis does not require complete runtime data but can still achieve most tasks.

3. The completely decoupled offline taint analysis allows StraightTaint to perform *full-featured* taint propagation strategies. The symbolic execution style taint propagation can accurately describe the intricate semantics of the x86 instructions, and also naturally support multi-tag and bit-level taint analysis.

4. We introduce a new concept, *Conditional Tainting*, based on the symbolic taint

analysis of straight-line code. Conditional tainting not only reports more precise and useful taint results but also opens many new important applications.

5. We propose a novel approach, TaintPipe, to efficiently decouple conventional inlined dynamic taint analysis by pipelining symbolic taint analysis on segments of straight-line code.

6. Unlike previous taint decoupling work, which suffers from frequent communication and synchronization, we demonstrate that with very lightweight runtime value logging, TaintPipe rivals conventional inlined dynamic taint analysis in precision.

7. Our approach does not require any specific hardware support or offline preprocessing, so both StraightTaint and TaintPipe are able to work on commodity hardware instantly and naturally multi-tag taint analysis methods.

8. We present BinCFP, an efficient multi-threaded binary code control flow profiling tool. We take advantage of pervasive multi-core platforms to greatly reduce the program execution slowdown. BinCFP's control flow profile does not require fine-grained static analysis so that BinCFP is fit for analyzing obfuscated binary code.

We also summarize the main benefits associated with our proposed decoupled taint analysis method.

1. Once a log is captured, it can be analyzed by StraightTaint multiple times. This feature is particularly useful when the exact analysis task is hard to anticipate. In our multi-tag taint propagation evaluation, we vary the number of taint tags in each round. StraightTaint only needs to log the required online data once and performs the multiple propagation rounds offline.

2. StraightTaint makes it possible to conduct *ex post facto* logging-based taint analysis in the cloud [39]. Service providers can deploy lightweight online logging in their services, and cloud hosts provide storage space for the logged data. Users can require a service to audit their sensitive data flow offline.

Chapter 3
Related Work

Taint analysis has been widely studied in various scenarios. Here we discuss the more related work along several axes. We first present the previous work on static and dynamic taint analysis. Our work is a hybrid of these two analyses. Then we introduce the efforts on taint logic code optimization, which are orthogonal to our approach and can benefit our symbolic taint analysis. Next, we describe the recent work on decoupling taint tracking logic from original program execution, which is the closest to our decoupled symbolic taint analysis. We also introduce related work on program replay and secure information flow. At last, since we perform symbolic execution style taint propagation on the straight-line code, we introduce the related work on dynamic symbolic execution as well.

3.1 Static Taint Analysis

Taint analysis comes in two major flavors: static and dynamic. Static taint analysis (STA) [11, 12, 40] has shown the capability of detecting data flows by considering all possible execution paths. Since static taint analysis (STA) is performed prior to execution by considering all possible execution paths, it does not affect application

runtime performance. STA has been applied to data lifetime analysis for Android applications [11, 41, 42], exploit code detection [12], and binary vulnerability exploration [40]. FlowDroid [11] can model the lifecycle of Android applications and conduct static data flow analysis. It uses on-demand alias analysis to achieve both precision and efficiency. Bodden et al. [41] proposed a generic IFDS/IDE solver. Static data flow analysis can be modeled as a graph reachability problem and be solved using the IFDS/IDE solver. Klieber et al. [42] proposed a static taint analysis technique which can identify data flows in individual Android applications, and discover dangerous data flows among different applications. However, STA may lead to either under-tainting or over-tainting [14] when merging two control flow paths. Furthermore, STA on obfuscated binary is quite challenging because even static disassembly of stripped binaries is still an unsolved problem [43].

3.2 Dynamic Taint Analysis

Dynamic taint analysis (DTA) is more precise than static taint analysis as it only propagates taint following the real path taken at run time. DTA has been widely used in various security applications, including data flow policy enforcement [4, 5, 16], reversing protocol data structures [44–46], malware analysis [47], and Android security [48]. In addition, DTA has also been used in binary difference analysis [37] and software plagiarism detection [49–54]. However, an intrinsic limitation of DTA is its significant performance slowdown. For example, TaintCheck [4] performs DTA based on dynamic binary instrumentation and imposes runtime slowdown as high as 40x. Various modern DTA techniques have been proposed. Source-level instrumentation techniques essentially compile the taint analysis code together with the program source code through source code rewriting [5, 6, 55, 56]. While they can achieve decent runtime performance, they

require the availability of source code, which is not applicable to legacy/proprietary applications and malware analysis. Hardware-assisted approaches [57, 58] associate every memory byte/register with a hardware tag to keep track of the taint information. Although they can achieve very good performance, their requirement of special hardware prevents them from being adopted in commodity hardware.

StraightTaint is a hybrid approach and therefore can potentially achieve higher precision than STA and better performance than DTA. Schwartz et al. [14] formally defined the operational semantics for DTA and forward symbolic execution (FSE). Our TaintPipe approach is in fact a combination of these techniques. For TaintPipe, the worker thread conducts concrete taint analysis (like DTA) whenever explicit taint information is available; otherwise symbolic taint analysis (like STA and FSE) is performed.

3.3 Taint Logic Optimization

Taint logic code, deciding whether and how to propagate taint, require additional instructions and "context switches". Frequently executing taint logic code incurs substantial overhead. Minemu [59] achieved a decent runtime performance at the cost of sacrificing memory space to speed up shadow memory access. Moreover, Minemu utilized spare SSE registers to alleviate the pressure of general register spilling. As a result, Minemu only worked on 32-bit program. TaintEraser [8] developed function summaries for Windows programs to propagate taint at function level. Libdft [19] introduced two guidelines to facilitate DBI's code inlining: 1) tag propagation code should have no branch; 2) shadow memory updates should be accomplished with a single assignment. Ruwase et al. [20] applied compiler optimization techniques to eliminate redundant taint logic code in hot paths. Jee et al. [60] proposed *Taint Flow Algebra* to summarize the semantics of taint logic for basic blocks.

3.4 Dynamic Taint Analysis Decoupling

A number of researchers have considered the high performance penalty imposed by inlined dynamic taint analysis. They proposed various solutions to decouple taint tracking logic from application under examination [22–26, 61], which are close in spirit to our proposed approach. Speck [22] forked multiple taint analysis processes from application execution to spare cores by means of speculative execution, and utilized record/replay to synchronize taint analysis processes. Speck required OS level support for speculative execution and rollback. Speck's approach sacrifices processing power to achieve acceleration. Similar to TaintPipe's segmented symbolic taint analysis, Ruwase et al. [23] proposed *symbolic inheritance tracking* to parallelize dynamic taint analysis. TaintPipe differs from Ruwase et al.'s approach in three ways: 1) Their approach was built on top of a log-based architecture [62] for efficient communication with idle cores, while TaintPipe works on commodity multi-core hardware directly. 2) To achieve better parallelization, they adopted a relaxed taint propagation policy to set a binary operation as untainted, while TaintPipe performs full-fledged taint propagation so that we provide stronger security guarantees. 3) They used a separate "master" processor to update each segment's taint status sequentially, while TaintPipe resolves symbolic taint states between two consecutive segments. Our TaintPipe could achieve better performance when there are more than a few "worker" processors.

Software-only approaches are the most related to StraightTaint and TaintPipe. They decouple dynamic taint analysis to a shadow thread by logging the runtime values that are needed for taint analysis. Two major categories have been proposed to decouple taint analysis from program execution. The first category, like TaintPipe, parallelizes dynamic taint analysis by delivering the needed runtime values to another core [22,23,25,26,30,63].

DECAF [30] extends Temu [64] to support asynchronous heavyweight taint propagation. However, DECAF does not show the performance gains introduced by its asynchronous tainting. Because of the strict synchronization requirement, most tools in the first category adopt incomplete taint propagation strategies to catch up the application execution.

The second direction, like StraightTaint, first records the application execution and then replay the taint analysis on a different CPU [27–29, 31, 65]. PIRATE [28] performs offline taint analysis on LLVM IR. It also first records an execution trace and then translates it to an intermediate language for offline taint analysis. However, PIRATE collects complete runtime information while StraightTaint only logs control flow profile. The most related work to StraightTaint is FlowWalker [27], which also uses Pin to record CPU context, and then performs a multi-tag assembly level taint propagation offline. However, StraightTaint reveals two distinct advantages. First, we design a more compact profile structure and multithreaded fast buffering scheme to parallelize the runtime data logging. Second, our offline taint analysis is performed on a side-effect free intermediate language instead of cumbersome x86 instructions. As demonstrated in our evaluation, StraightTaint outperforms FlowWalker with better performance and accuracy.

Recent work ShadowReplica [26] ameliorates this drawback by adopting fine-grained offline optimizations to remove redundant taint logic code. In principle, it is possible to remove redundant taint logic by means of static offline optimizations. Unfortunately, even static disassembly of stripped binaries is still a challenge [66, 67]. Therefore, the assumption by ShadowReplica that an accurate control flow graph can be constructed may not be feasible in certain scenarios, such as analyzing control flow obfuscated software. We take a different angle to address this issue with lightweight runtime information logging and segmented symbolic taint analysis. We demonstrate the capability of StraightTaint and TaintPipe in speeding up obfuscated binary analysis,

which ShadowReplica may not be able to handle. Furthermore, ShadowReplica does not support bit-level and multi-tag taint analysis, while StraightTaint and TaintPipe handle them naturally.

3.5 System and Program Replay

System replay and program replay techniques can precisely repeat an execution and allow analysts to conduct post analysis using dynamic taint analysis tools. Aftersight [68] and ReEmu [69] record nondeterministic virtual machine inputs and can replay the execution or dynamic analysis. Analogously, PinPlay [70] supports user-level deterministic replay. Our approach does not rely on complete execution reply. We only log executed control flow information to recover the straight-line trace later.

3.6 Secure Information Flow

There have been much research interests in securing information flow through enforcing information flow policies based on specifications about information flow. Merlin [71] is a tool that can statically infer and generate specifications about information flow of web applications. Based on the information flow specifications, information flow policies can be enforced and vulnerabilities can be detected. JFlow [72] is an extension to Java language, which support statically-checked information flow annotations. King et al. [73] proposed a model to discover the source of information flow errors in source code. However, these approaches without exception requires source code. In contrast, our techniques work with binary.

3.7 Dynamic Symbolic Execution

Another related area to our symbolic taint analysis is dynamic symbolic execution, namely *concolic testing* [74–77], a method of combining concrete execution with symbolic execution. Our method is similar to the concolic testing in that we map symbols to taint seeds and then perform the symbolic taint analysis along a recorded execution trace. Also, our method can benefit from symbolic execution optimization work to speed up taint analysis, such as memoized symbolic execution [78]. However, we have different goals. Dynamic symbolic execution is mainly for automatic input generation to explore more paths while our primary interest lies in accurate taint analysis on the straight-line code. In addition, concolic testing relies on complete runtime information while our symbolic taint analysis only depends on limited runtime information. The recent work, Hercules [79], also mentions the idea of using symbolic execution for precise taint tracking. However, our approach has a strikingly different purpose with Hercules. Hercules is for reproducing crashes in benign application binaries; while StraightTaint and TaintPipe are designed to speed up reverse engineering tasks on binary code.

Chapter 4

StraightTaint: Decoupled Offline Symbolic Taint Analysis

The previous work on dynamic taint analysis decoupling seeks to offload taint analysis from program execution and run it on a spare core or a different CPU. However, since the taint analysis has heavy data and control dependencies on the program execution, the massive data in recording and transformation overshadow the benefit of decoupling. In this chapter, we propose a novel technique to allow very lightweight logging, resulting in much lower execution slowdown, while still permitting us to perform full-featured offline taint analysis, including bit-level and multi-tag taint analysis. We develop StraightTaint [65], a hybrid taint analysis tool that completely decouples the program execution and taint analysis. StraightTaint relies on very lightweight logging of the execution information to reconstruct a straight-line code, enabling an offline symbolic taint analysis without frequent data communication with the application. While StraightTaint does not log complete runtime or input values, it is able to precisely identify the causal relationships between sources and sinks. Compared with traditional dynamic taint analysis tools, StraightTaint has much lower application runtime overhead.

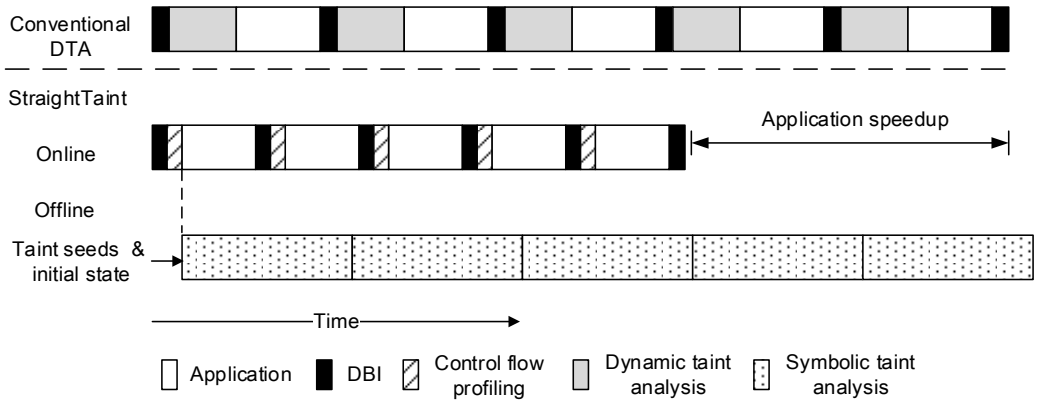

Figure 4.1. Conventional DTA vs. StraightTaint.

4.1 Background and Overview

In this section, we first discuss the drawbacks of current dynamic taint analysis techniques. This also inspires us to propose our method. We describe the limitations of the previous work with a motivating example. We show C code for ease of understanding even though StraightTaint works on the binary code. At last, we introduce the architecture of StraightTaint and its components.

4.1.1 Dynamic Taint Analysis Optimization

Typically in dynamic taint analysis (DTA), the data derived from untrusted sources are labeled as *tainted* (i.e., taint seeds). The propagation of the tainted data will be tracked as the program executes according to the taint propagation policy. Then the taint status will be checked at certain critical locations (i.e., taint sinks). DTA has been broadly employed in security applications. However, an inherent limitation of conventional DTA is that taint logic is strictly coupled with program execution. Figure 4.1 illustrates a conventional DTA tool built on dynamic binary instrumentation (DBI). The taint tracking code is interleaved with program execution, leading to frequent context switches and resource competitions

between the application and analysis code. As a result, the application under examination is significantly slowed down. Various advanced DTA techniques have been proposed to achieve decent runtime performance [59, 80]. Unfortunately, they either rely on an ad hoc emulator [59] or cannot work on commodity hardware [80]. Decoupling taint analysis from program execution has been demonstrated as an effective approach. However, due to the heavy data and control-flow dependencies on the application execution, decoupled taint analysis cannot run independently. Intuitively, each memory address and control transfer target have to be delivered to the decoupled taint analysis. Therefore, the large logged data is a barrier to further improving the performance.

As shown in Figure 4.1, our key insight is that taint analysis can be completely decoupled from program execution, without frequent online communication and synchronization. Offline taint analysis can be performed based on control flow information and very little runtime data (e.g., the initial execution state at a taint source). We notice that memory reference operations in x86 architecture are addressed through registers and constant offset calculations. For example, `mov ebx [4*eax+4]` loads the content stored at the address `4*eax+4` to `ebx`. With the initial execution state and the straight-line code, most memory reference addresses can be recovered. The proposed StraightTaint explores this idea.

Note that the deterministic replay work [47, 70, 81], which records non-deterministic inputs and replays them on an offline analysis, can be applied to decouple taint analysis as well. Compared to StraightTaint, the logs are smaller, and the online performance could be better. However, the logged data contains little information about execution, making it impossible for direct taint analysis. Furthermore, the offline replay overhead is quite high. For example, Aftersight [81] replays a single-tag taint analysis on a QEMU-based CPU simulator, but the performance is as high as 100X slowdown. Our solution represents a

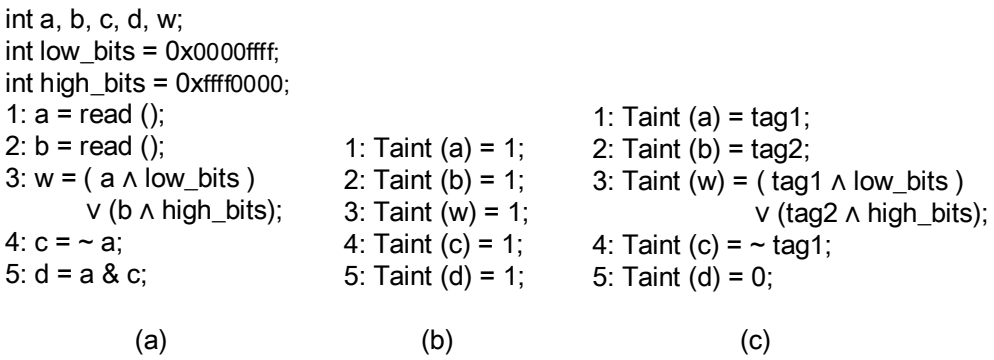

Figure 4.2. Conventional single-tag taint propagation vs. StraightTaint multi-tag symbolic taint propagation: (a) a sequence arithmetic operations; (b) conventional single-tag taint propagation results; (c) StraightTaint multi-tag symbolic taint propagation results.

middle ground that balances the performance between online logging and offline taint analysis.

4.1.2 Incomplete Taint Propagation Strategies

As conventional DTA tools are subject to limited computing resources, typically they have to adopt incomplete taint propagation strategies to achieve an acceptable performance. In many cases, such conservative strategies lead to the precision loss. The first drawback comes from the single-tag propagation. Most DTA tools associate each variable with one shadow memory bit or byte to represent the taint status: 1 means tainted and 0 means untainted. Although single-tag can work in some simple scenarios, multi-tag taint analysis has much broader security applications. For example, BitFuzz [9] assigns different taint tags to input bytes and then detects encoding functions in malware by identifying high taint degree; iBinHunt [37] utilize multi-tag taint analysis to reduce the number of possible basic blocks to compare. Furthermore, many arithmetic and logic operation results overlap the operands so that a taint tag may come from multiple sources. Therefore, the multi-tag attribute is essential for accuracy as well. The second limitation

is due to the conservative propagation strategies when dealing with the complicated x86 instructions. These simple strategies are fast but neglect the particular instruction semantics that may affect the taint propagation results. In addition to the frequently used "short circuiting" solution, some previous work tracks the taint flow only through unary operations (the output of a binary operation is set as untainted) to achieve better parallelization [23].

Figure 4.2 presents a snippet of an encoding function, which is frequently used in malware [9]. Figure 4.2 (a) lists a straight-line code with complicated arithmetic operations. Conventional DTA performs the taint analysis on this code snippet with single-tag and "short circuiting" strategies. Figure 4.2 (b) shows the propagation results: all variables are tainted. Look carefully at line 3 in Figure 4.2 (a), the taint tag of variable `w` derives from two taint seeds but conventional DTA just labels it as a single tag. Besides, the variable `d` will always be zero because `c` is the `bitwise NOT` of `a`. However, the "short circuiting" propagation mistakenly label `d` as tainted, resulting in over-tainting [14]. A nature benefit of StraightTaint's offline taint analysis is that supporting full-featured taint propagation strategies is straightforward, such as multi-tag and bit-level taint analysis. Also, our symbolic taint analysis on the straight-line code can capture intricate details of the x86 instructions. Figure 4.2 (c) shows the results of StraightTaint multi-tag symbolic taint propagation: `w` and `c` are correctly tainted; the taint tags of `d` are cleaned as expected. StraightTaint avoids the imprecision and over-tainting problems introduced by previous incomplete taint propagation strategies.

4.1.3 Architecture

Figure 4.3 illustrates the architecture of StraightTaint, which consists of two stages: online logging and offline analysis. The first stage, as shown in the left part of Figure 4.3,

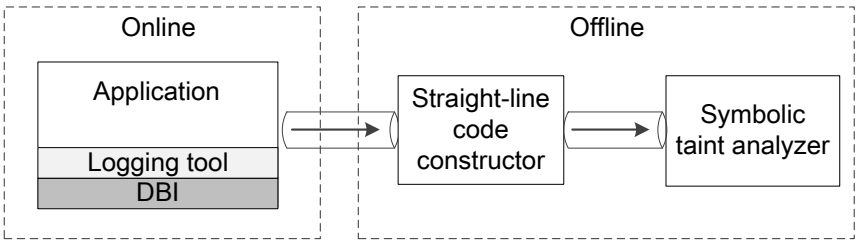

Figure 4.3. The architecture of StraightTaint.

involves very lightweight online logging to mainly record control flow information. We built a logging tool using dynamic binary instrumentation (DBI), enabling StraightTaint to work with unmodified program binaries directly. Application thread(s) are executing over the DBI and our logging tool. Our logging tool dynamically instruments each executed basic block to record the execution using tags that are unique for each basic block. The basic block tags are written to a trace buffer and then stored to a disk storage when the buffer is filled up. Careful design of the online logging tool is crucial for achieving better efficiency. Therefore, we propose three guidelines and the details will be discussed in Section 4.2.

The generated log data is passed to the offline taint analysis (the right component of Figure 4.3). This stage first reconstructs the straight-line code trace from the log data, and then lifts the x86 instructions to BIL [33], an RISC-like intermediate language. The core of our symbolic taint analyzer is an abstract taint analysis processor. Similar to the shadow memory in DTA, StraightTaint maintains a *context* structure to store symbolic taint variables and concrete values. Our offline taint analyzer is able to carry out both forward taint tracking to detect the effect of an intrusion, and backward tracing to identify attack provenance. Even without complete runtime data information, StraightTaint can achieve comparable precision as dynamic taint analysis, which will be discussed in detail in Section 4.3.

4.2 Efficient Online Logging

StraightTaint applies a lightweight logging to lower the impact on application performance. Since not all the instructions executed are of interest, we invoke online trace logging when pre-defined taint seeds are first introduced. In StraightTaint, a user can set the input data from keyboard, file, network or function return value as taint seeds. To avoid symbolic taint variables explosion in the offline analysis, we leverage the concrete execution state when the taint seeds are introduced to constrain fresh symbolic taint variables. We collect an execution state by a process dump of our logging tool. Besides the initial execution state, only the executed control flow information is logged to reconstruct the straight-line code later.

The logged data are first stored in a memory buffer and then dumped to disk storage when the buffer is filled up. Three design goals guide us to achieving low online logging overhead: 1) the logged data representation should be compact so that trace buffer holds as much data as possible before it becomes full; 2) application thread(s) (i.e. producer) should not be blocked when the full buffers are being consumed, that is, processing the buffers asynchronously; 3) application instrumentation overhead should be minimized. We meet the first requirement by extending an advanced trace profiling format [82]. To address the second challenge, we propose an n-way fast buffering scheme by exploiting multi-cores to parallelize profile consumption. At last, we carefully design our instrumentation code to favor code inlining and avoid frequent context switches. Figure 4.4 illustrates the overview of StraightTaint's online logging part, which consists of three parts: online logging, multi-threaded fast buffering, and straight-line code reconstruction.

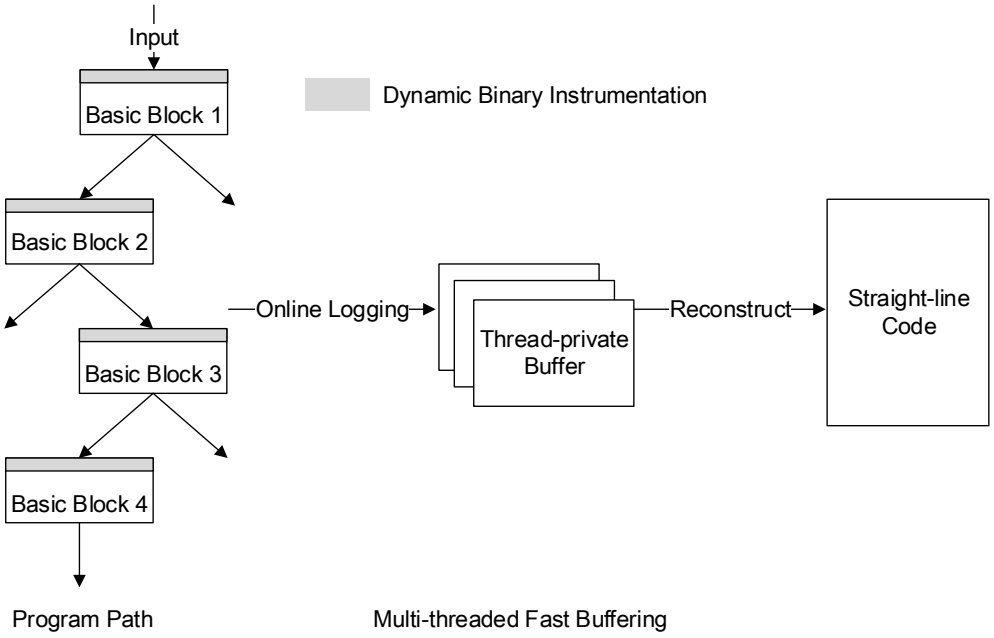

Figure 4.4. Online logging overview.

4.2.1 Control Flow Profile

Since dynamic control flow information can be represented as a sequence of basic blocks executed, a naive approach is to record each basic block's entry address. On a 32-bit machine, a 4-byte tag is needed to label a basic block. However, a full 4-byte tag is an excessive use and would take up too much space. Zhao et al. [82] proposed the *Detailed Execution Profile* (DEP), an efficient method that only uses 2-byte tags to record most of the basic blocks and handles special cases with extra escape bytes. DEP splits a 4-byte address into 2 high bytes for *H-tag* and 2 low bytes for *L-tag*. During control flow profiling, if two sequential basic blocks share the same H-tag, only L-tag of each basic block is logged into the profile buffer. If the two H-tags are different, a special escape tag 0x0000 followed by the new H-tag will be entered into the buffer. Furthermore, DEP's scheme does not require control flow graph or any fine-grained static analysis, making it suitable for binary code analysis. Our approach improves DEP's scheme in several ways.

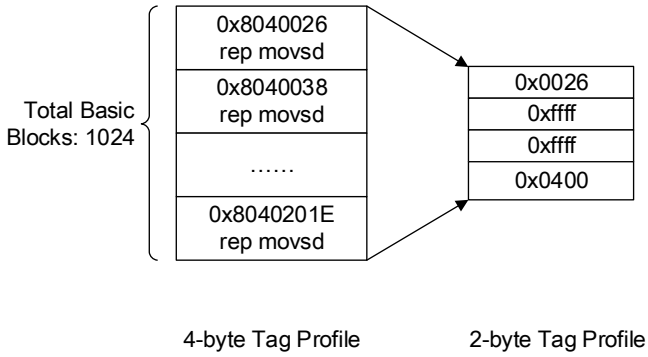

Figure 4.5. Optimization to the single basic blocks caused by REP-prefixed instructions.

4.2.2 Optimizations

Certain x86 instructions are related to string operations, such as `MOVS`, `LODS`, `STOS`, `CMPS` and `SCAS`. These instructions with `REP`-prefix are typically executed repeatedly. Dynamic binary instrumentation tools [17, 83] usually treat `REP`-prefixed instructions as implicit loops. If a `REP`-prefixed instruction iterates more than once, each new iteration will cause a single instruction basic block to be generated. In such cases we'll see much more basic blocks than we expect. In SPEC CPU2006, we find several cases that `REP`-prefixed string instructions are heavily used. For example, as high as 13.4% of total executed instructions of the `h264ref` benchmark program are `REP` instruction repetitions. StraightTaint extends DEP's profile to handle implicit loops introduced by the `REP`-prefixed instructions, which could otherwise become a performance overkill. Particularly, we inspect the first loop of `REP`-prefixed instructions. And then two consecutive escape value 0xffff, followed by a repeat count (stored in the `ecx` register) are entered into the trace buffer to represent the entire implicit loops of `REP`-prefixed instructions. The maximum `REP`-prefixed loop count we see in our evaluation comes from `gcc` benchmark, which is $1,770$, far less than two-byte number limit. Figure 5.4 presents such an example. The left part shows a total $1,024$ single instruction basic

blocks due to the implicit loop unrolling. StraightTaint only encodes the first repetition and appends two consecutive 0xffff to indicate such `REP` loops. After that, the number of repetitions (0x0400 in our example) is put into the profile buffer. Consequently, StraightTaint only consumes 8 bytes to represent the total 1024 basic blocks. In contrast, the raw 4-byte profile takes up 1024×4 bytes space and DEP needs 1024×2 bytes.

Furthermore, we also configure Pin to disable unrolling `REP`-prefixed instructions as Pin otherwise inserts analysis code into each implicit iteration, introducing additional overhead. In our evaluation, our optimization can reduce DEP profile space at most 55.2% and with up to 63.5% runtime overhead reduction. Noth that it is also possible to use a single bit to log a basic block by recording the binary decision of conditional jump [84], which leads to a much denser log data. However, Pin's notion of basic blocks is not strictly the same as compiler-level basic blocks. Pin also breaks basic blocks on some specific instructions, for instance (on IA-32 and IA-64 architectures) `CPUID`, `POPF` and `REP`-prefixed instructions. Single bit logging may miss such special cases. Also, unlike StraightTaint,recovering the whole execution trace from single bit profile has to walk through the control flow graph. Although our example is 2-byte tag profile on IA-32, it is straightforward to extend to 4-byte tag profile for 64-bit machines. Current StraightTaint accommodates both 32-bit and 64-bit binaries. It is worth mentioning that frequently checking of H-tags during online logging can introduce large additional overhead. We will further discuss this issue in Section 4.2.4.

4.2.3 Multithreaded Fast Buffering Scheme

In this section, we focus on the design of a general scheme that supports concurrently buffering data of multi-threads on the multi-cores platform. The goal is to exploit underutilized computing resources to alleviate the disk I/O bottleneck. At the center of

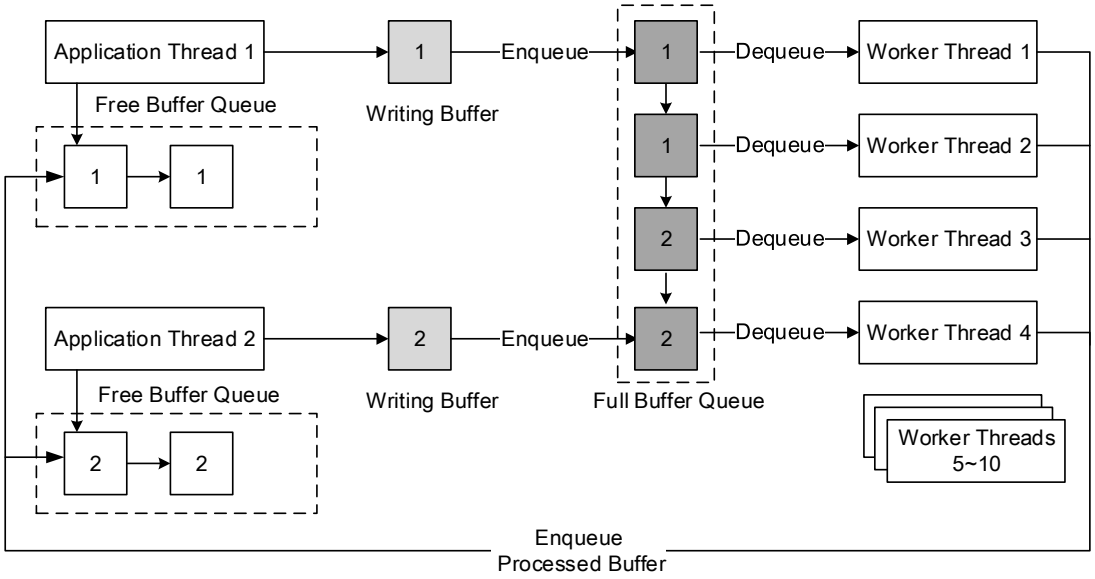

Figure 4.6. Multi-threaded fast buffering scheme.

our design is a buffering thread pool, in which multiple buffers enable the application instrumented to continue executing and filling up free buffers, while multiple Pin-tool internal spawned threads process full buffers asynchronously. Figure 4.6 illustrates how fast buffering thread pool works. Let us assume the application under examination contains two threads (i.e., *producer threads*). The processing steps of our fast buffering scheme are as follows.

1. When a program starts running, each application thread allocates a number of thread-private buffers (5 buffers in our example), which means the instrumented code of each thread can only write data into its private buffers. Available free buffers are formed as a free-buffer queue. Also, each free buffer contains a thread ID, indicating the associated application thread of the free buffer.

2. Simultaneously, several of Pin-tool internal threads are spawned as well. We call them *worker threads* (10 worker threads in our example). Note that different from application threads, worker threads are not JITed and can execute natively. A

worker thread takes a buffer from the full buffer queue and then dumps the buffer data to disk storage. After that, the worker thread put the buffer into the free buffer queue of the associated application thread. Multiple worker threads access a full buffer exclusively by acquiring the buffer's lock.

3. Worker threads communicate with application threads via counting semaphores. After being created, worker threads are waiting for the arrival of full buffers. The application thread first fills one free buffer. When this buffer becomes full, a callback function, *BufferFull* will be called to perform two tasks: 1) enqueues the full buffer to the global full-buffer queue and wakes up one worker thread to process it; 2) returns the next available thread-private free buffer to the application thread. If the free-buffer queue is empty, the application thread has to be blocked until a free buffer is available.

It is apparent that the less time application threads spend on waiting for free buffers, the better performance it would achieve. Therefore, we bias the implementation of our fast buffering scheme to favor application runtime performance. Specifically, we create enough worker threads to ensure a full buffer can be processed immediately. We can also dynamically adjust the number of buffers and worker threads to optimize the synchronization and load balancing. In Figure 4.6, the number of worker threads is 10, equal to the total number of buffers allocated by application threads. It is evident that the availability of unused cores and the size of profile buffer have a great impact on the runtime overhead. In Section 4.5.1, we will discuss how to tune these two factors.

4.2.4 Instrumentation Optimization

StraightTaint's online logging tool is built on Pin. The way a Pin tool is implemented can have great impact on the performance of the instrumented application. Besides the

optimization to the `REP`-prefixed instructions, this section introduces other methods we applied to reduce StraightTaint's instrumentation overhead.

Conceptually, Pin's instrumentation contains two major components, namely instrumentation code and analysis code. Instrumentation code inspects program to decide where the analysis code should be injected. In our tool, the instrumentation granularity is on the basic block level. The instrumentation code is executed only once for every sequence of basic blocks, and the translated code, including original code and analysis code, is saved in Pin's code cache for efficiency. Analysis code will be invoked at beginning of each basic block. In our case, the analysis routines record dynamic control flow and manage fast buffering scheme. As analysis routines are executed very frequently, carefully tuning the analysis code is paramount for better performance. Pin tends to inline the compact, branch-less analysis routines into translated code cache; while the analysis code with conditional branches will force Pin to emit a function call to the respective analysis routines instead. In our case, we have to frequently check whether the H-tag has changed upon the execution of each basic block and write the tag to the trace buffer accordingly. To favor Pin's inlining, we shift the check of H-tag to the instrumentation phrase, which is performed only once when each basic block is translated. In this way, In this way, the branch instructions in the analysis routines are removed while profile tags are updated as well. As a result, the overhead introduced by frequently checking H-tags is reduced. In Section 4.4, we will introduce other Pin-specific optimizations we adopted.

4.3 Offline Symbolic Taint Analysis

4.3.1 Reconstruction of Straight-line Code

Given the trace collected from the online logging, reconstructing a complete sequence of 4-byte starting addresses of basic blocks is quite straightforward. The beginning of the trace profile should be a special value 0x0000, followed by an H-tag. Each basic block 4-byte entry address is the concatenation of its corresponding H-tag and L-tag. Then the x86 instructions of each basic block are extracted from the application's disassembly code. An elaborate knowledge of the x86 ISA is required to accurately track taint propagation that operates at the binary level. However, the cumbersome x86 ISA makes it an extremely tedious engineering work. For example, previous work such as libdft [19] contains more than $5,000$ lines of code to handle the x86 ISA complexity. Figure 4.2(a) shows an example involving complicated arithmetic operations. Furthermore, some x86 instructions contains implicit side effects, which only propagate taint conditionally according to the contents of EFLAGS. For example, `CMOVcc` instructions propagate a taint tag only if the EFLAGS register flags are in a specified state.

To reduce engineering workload and simplify taint tracking, we lift up x86 instructions to BIL [33], an RISC-like intermediate language (IL). IL models each x86 instruction semantics with one or several IL statements, which is able to get rid of the intricate details of the x86 ISA, leaving us only 25 IL statements that we need to analyze carefully for accurate taint tracking. In our evaluation, we have demonstrated that our IL-style taint analysis is more accurate than the previous work with with incomplete taint propagation strategies. Note that with control flow information, we have resolved each indirect jump target and direction of conditional jump in the straight-line IL code.

4.3.2 Symbolic Taint Analysis

By labeling the stream bytes of taint seeds as symbolic variables, StraightTaint offline taint propagation is a kind of symbolic execution on the straight-line code. Since each taint seed byte can be associated with a fresh symbol, multi-tag taint propagation is natural for StraightTaint. The core of our symbolic taint analysis engine (as shown in Figure 4.7) is an abstract processor, which maintains a *context* structure as the execution state. The context structure consists of a program counter pc, a variable context V and a memory context M. For conciseness, we represent the state of the abstract processor with the tuple $s = (pc, V, M)$. The variable context V contains all symbolic register values (e.g., general purpose registers and bits of EFLAGS) and temporaries. The temporaries are the expressions used in the static single assignment form of BIL. We also explicitly represent the return value of a function as a special variable to facilitate detecting buffer overflow vulnerabilities. The memory context M, with a structure analogous to the two-level architecture of x86 virtual addressing, is a mapping from memory addresses to their symbolic variables. By interpreting the current IL at pc, a state of the abstract processor $s = (pc, V, M)$ is translated into a new state $s' = (pc', V', M')$ and V' and M' are updated according to the semantic of the IL. At the same time, StraightTaint checks whether a location of interest (i.e., taint sink) is tainted by checking whether its value is a symbolic expression. After the last IL is simulated, pc is set to `halt` and V and M are not updated anymore.

We start offline taint analysis when the pre-defined taint seeds are first introduced to the application. Besides the taint seeds, there could be other uninitialized variables such as the stack pointer and memory contents. In principle, we can assign a fresh symbolic variable to each uninitialized variable. However, symbolic taint analysis with an unconstrained initial state can quickly reach the memory capacity and lead

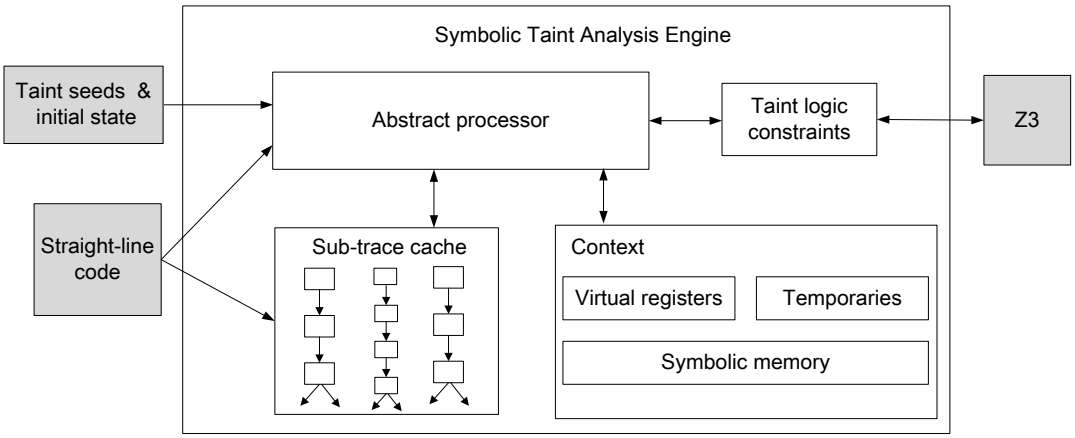

Figure 4.7. Symbolic taint analysis engine.

to the problem of "over-tainting" [14] as well. Our solution is to leverage a process dump to assign other uninitialized variables with concrete values, only leaving the taint seeds as symbolic variables. Here we use another tricky example to show the value of symbolic execution style taint propagation. To reduce the number of conditional jumps, some compiler optimization options translate conditional instructions into a sequence of arithmetic operations. Figure 4.8 (a) shows such an example we find in our test cases. Figure 4.8 (b) lists the semantics for each instruction. The net result of the sequence of arithmetic operations is presented in Figure 4.8 (c), which is a branch condition. The taint tag of `reg` is either from `val1` or `val2`. StraightTaint successfully propagate taint tags for this tricky case, while the other DTA tools such as Temu [64], Dytan [15], and libdft [19] all fail.

4.3.3 Memory Reference Address Resolution

Another issue for StraightTaint's offline taint analysis is that we do not record memory reference addresses, which are typically calculated through general registers and constant offsets. Our observation is that, with the initial execution state and the straight-line code, most memory reference addresses are computable along the symbolic taint analysis.

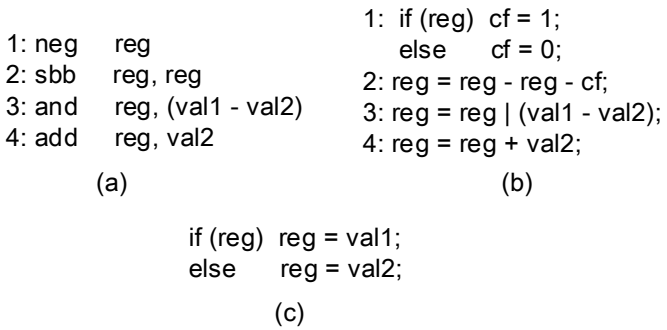

Figure 4.8. Example: branchless logic code (`reg` stands for register; `val1` and `val2` are two tainted variables).

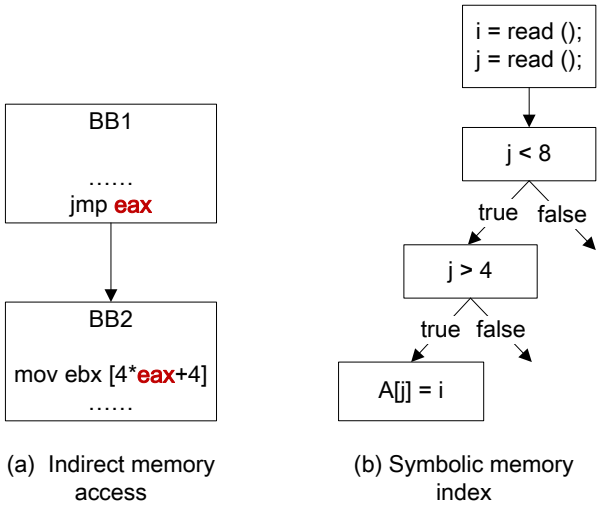

Figure 4.9. Example: memory reference address resolution.

Figure 4.9 (a) shows how we resolve an indirect memory access. Since we have resolved each indirect jump target in the straight-line code (See Section 4.3.1), the memory indirect access through `eax` in `BB2` can be determined. To solve a memory address `address_a` that is cannot be computed accurately (e.g., heap memory allocation), we allocate memory on-the-fly. Inspired by micro execution [85], we use return value of `malloc(1)` as `address_a`, which guarantees that `address_a` would not conflict with an existing address. A symbolic index happens when a symbolic variable is used as the index of a memory lookup. Intuitively a symbolic memory index could point to

any memory slot. We deal with this problem by solving path conditions. As shown in Figure 4.9 (b), the path conditions along the straight-line code restrict the range of symbolic memory index `j` within $4 < j < 8$. Then we conservatively label all the possible memory values as tainted. For the example in Figure 4.9 (b), A[5], A[6], and A[7] will be tainted.

4.3.4 Conditional Tainting

As x86 conditional control transfer instructions typically depend on the value of the `EFLAGS` register (e.g., jz and jo) our virtual registers also keep track of bit-level symbolic variables for `EFLAGS`. When a symbolic expression is used in a conditional jump instruction, we collect it as a branch condition. After a complete symbolic taint propagation run, the conjunction of all branch conditions is the *Taint Logic Constraints*. Thus, the values that satisfy the taint logic constraints are the concrete taint seeds that would lead the program to execute the same taint tracking operations as the one symbolically tainted. With taint logic constraints, which are solved by a theorem prover (e.g., Z3 [86]), previously taint analysis results can be mapped to new inputs and runtime values without DTA again!

4.3.5 Optimization

Like Pin's block cache to save the overhead of frequently executed basic block retranslation, we take a similar approach to speed up our offline symbolic taint analysis. We call it "sub-trace cache" (see "sub-trace cache" component in Figure 4.7). We merge sequential basic blocks that have one predecessor and one successor as a sub-trace, which can be viewed as an enlarged basic block. We represent the input-output relations of a sub-trace as a set of symbolic formulas and maintain a lookup table in the sub-trace

unit. Therefore, the successive runs can directly reuse previous results, without the need for recomputing them. Another primary optimization we adopt is function summary. Most well-known library functions have explicit semantics (e.g., C strings manipulation functions defined in string.h), and many of them even do not affect taint propagation (e.g., `strcmp`). Therefore, we turn off symbolic taint analysis at the boundary of these functions and update context according to their semantics summaries. For a sequence of adjacent memory access introduced by `REP`-prefixed instructions, we recover the number of repetitions from trace profile and perform batch processing instead of byte by byte operation.

4.4 Implementation

To demonstrate the idea of StraightTaint, we implemented a prototype including online logging based on the Pin DBI framework [17] (version 2.12) with $2,660$ lines of code in C/C++, and offline symbolic taint analysis engine on top of BAP [33] (version 0.8) with 4,540 lines of OCaml code. We rely on BAP to convert assembly instructions to IL and convert IL expressions to CVC formulas. We use Z3 [86] as our constraint solver. The saving and loading of sub-trace cache lookup table are implemented using the OCaml Marshal API, which encodes arbitrary data structures as sequences of bytes and then store them in a disk file.

When implementing the Pin-tool, we create thread local storage (TLS) slot to store and retrieve per-thread buffer structure. Note that Pin-tools are unable work with either pthreads library or Win32 threading API. We utilize the Pin thread API to spawn worker threads and implement a counting semaphore using Pin's own binary semaphore. To make the best of Pin's code cache effect, we enlarge the maximum number of basic blocks per Pin trace from 3 to 8. We also use GCC's built-in macro "`__builtin_expect()`" to

provide the compiler with the branch prediction. Furthermore, we perform low-overhead buffering of data through Pin's fast buffering APIs, which support inlining a callback function when a buffer becomes full. We also force Pin to use fastcall calling convention to pass arguments via registers to avoid emitting stack access instructions (i.e., `push` and `pop`).

4.5 Evaluation

Our testbed contains two machines. One is a server machine, which is equipped with two Intel Xeon E5-2690 processors (16-core with 2.9GHz) and 128GB of RAM. Another is a desktop, consisting of Intel Core i7-3770 processor (Quad Core with 3.40GHz) and 8GB memory. Both are running Ubuntu 12.04. The data presented throughout this section are all mean values. We calculate them by running five repetitions of each experiment case.

4.5.1 Configuration of Buffer Size and Worker Threads

We studied two factors that may affect StraightTaint online logging performance: 1) the buffer size of control flow profile; 2) the number of available worker threads. We first survey the impact of various buffer size. In order to achieve enough parallelism, the number of worker threads is set to 16 and 4, respectively. The total buffer sizes are therefore the number of worker threads $\times$ single buffer size. We choose SPEC CPU2006 with *test* workload as the training set. As shown in Figure 4.10, roughly the overhead decreases as the buffer size is increased. This is mainly due to the reduction of free/full buffer switches, and worker threads spend less time on synchronization. As the buffer size is beyond a certain point (64MB for the 16-core system and 128MB for the 4-core system), the slowdown is increased a little. We attribute this to the large total buffer

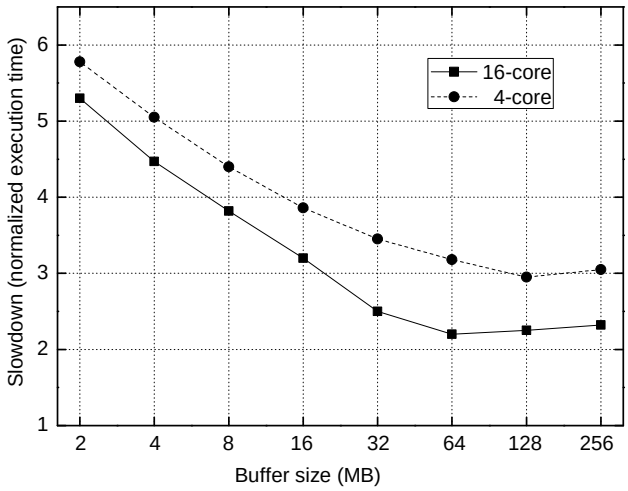

Figure 4.10. Normalized slowdown on 16-core and 4-core systems when profile buffer size varies.

sizes (e.g., 16 $\times$ 256MB) interfering with application's working set. Then we fix the buffer size to 64MB for the 16-core system and 128MB for the 4-core system and alter the number of worker threads. In general, the performance is better as more worker threads are added. Due to the maximum parallelism and the tuned buffer size, 16 worker threads with 64MB buffer size achieve the optimum result. We set these two parameters as default configuration and conduct the following experiments on the 16-core system.

4.5.2 StraightTaint vs. libdft

We first compare StraightTaint with libdft [19], a state-of-the-art inlined DTA tool built on Pin ("libdft" bar). In aid of evaluating the application performance slowdown imposed exclusively by StraightTaint, we develop a simple tool (nullpin) to measure Pin's environment runtime overhead, which runs a program under Pin without any form of analysis ("nullpin" bar). We also measure the logging overhead without buffering the profile data to disk ("online-no I/O" bar). Under this configuration, the application never stalls to wait for free profile buffers, which can represent the upper bound of performance

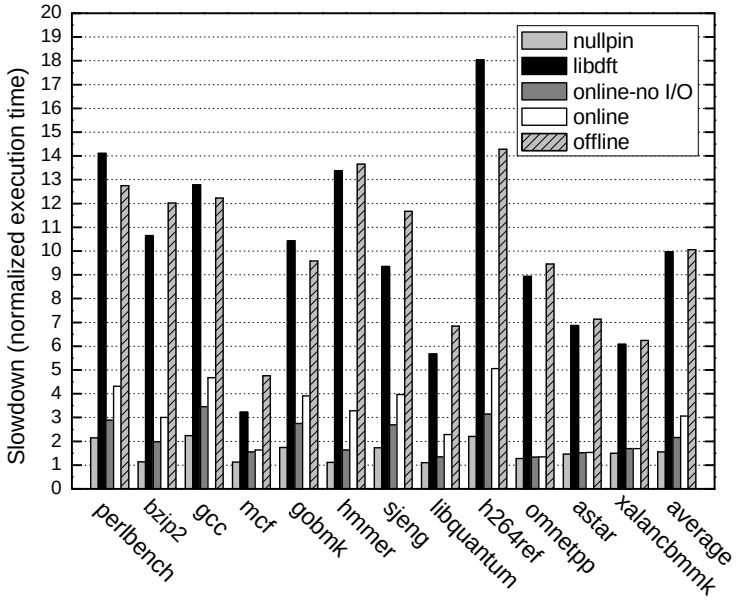

Figure 4.11. StraightTaint slowdown on SPEC CPU2006 (Y axes shows the normalized execution time).

improvement attainable by StraightTaint. Viewed from a different angle, "online-no I/O" bar also indicates the overhead introduced by Pin's instrumentation. All runtime data[1] presented in this section are normalized to application native execution time (without running Pin).

Figure 4.11 shows the normalized execution time of running SPEC CPU2006 int benchmark suite with *reference* workload. Since the reference workload is CPU-intensive, we expect that these results can estimate the worst case scenarios. On average, StraightTaint's online logging exhibits a 3.06X slowdown to native execution, while libdft lags behind as much as 9.96X, indicating that StraightTaint speeds up application execution by a factor of 3.25. It is noteworthy that if taking *nullpin* as the baseline, the slowdown exclusively introduced by StraightTaint is only 1.97X while for libdft is 6.43X. This number is in line with the observations by the previous work [20, 21]; that is,

[1]The "online" bar is calculated by counting wall-clock time because we have to consider the I/O time introduced by our buffering scheme. Other bars are calculated by counting CPU time.

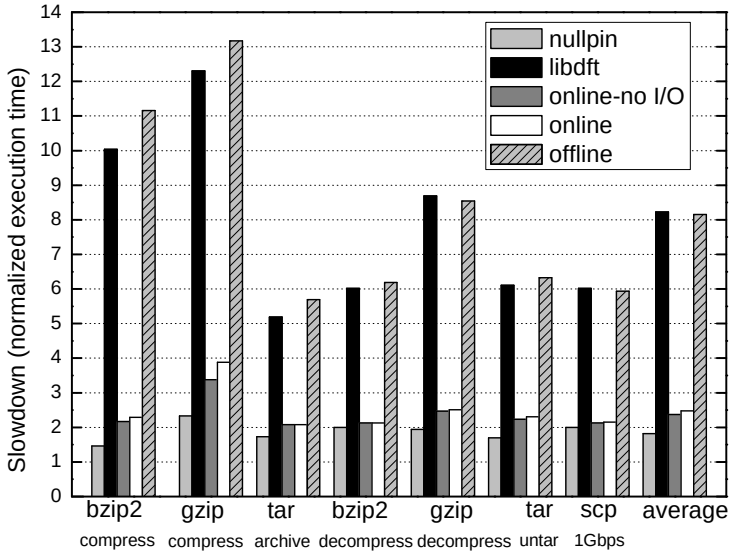

Figure 4.12. StraightTaint slowdown on common Linux utilities (Y axes shows the normalized execution time).

performing one taint propagation operation normally needs six extra instructions. The overhead incurred by StraightTaint's online instrumentation is 2.16X ("online-no I/O" bar), compared to Pin's environment runtime overhead (1.55X), 39.4% extra performance penalty added. Due to the CPU bounded test suite, StraightTaint has to put more efforts to deal with large amount of I/O. Therefore, additional 41.7% overhead to "online-without I/O" version is introduced.

Next we evaluated StraightTaint on four common Linux utilities that represent three kinds of workload. `tar` is I/O bounded, whereas `bzip2` and `gzip` are CPU intensive program. *scp* represents a middle level between these two cases. We use `tar` to archive and extract GNU Core utilities 8.13 package (∼50MB). And then we apply `bzip2` and `gzip` to compress and decompress the archive file of Core utilities. For *scp*, we copy the archive file of Core utilities over 1Gbps link. We achieve a similar improvement with the SPEC CPU2006 experiment. As shown in Figure 4.12, StraightTaint imposes a average 2.48X slowdown to native execution, with a 3.32 times speed up to libdft. Besides, the

additional overhead due to I/O effect is quite small (< 5%). We attribute this to our fast buffering scheme, which has significantly mitigated the bottleneck of I/O.

On average, StraightTaint generates about 2.8GB of raw trace profiling data for SPEC2006's reference workload. Compared to the raw 4-byte tag profile size, the relative size of StraightTaint is only 49.2%. In general, StraightTaint outperforms DEP's encoding [82] by 5 percentages in terms of smaller profiling data size. It is worth mentioning that we see a significant size reduction for the `h264ref` benchmark, from DEP's 4.8GB to 2.1GB. The reason is `h264ref` intensively utilizes `REP`-prefixed instructions, which are very well handled by StraightTaint's optimization.

The last bar for each application in Figure 4.11 presents the performance of symbolic taint analysis, which is normalized to native execution as well. Since we have decoupled taint tracking from program execution, offline symbolic taint analysis avoids the overhead introduced by DBI's environment and computing resource competitions. On the other hand, symbolic taint analysis engine is in fact an interpreter for each IL, which is much slower than native execution. To alleviate this issue, we have applied a number of optimization methods (discussed in Section 4.3.5). The net result is that our offline symbolic taint analysis takes approximately the same amount of time as libdft (10.06X for StraightTaint and 9.96X for libdft on average). In several cases (e.g., perlbench and h264ref), StraightTaint's offline part outperforms libdft. Considering that StraightTaint is aiming to shift dynamic taint analysis cost to the offline analysis phase, this degree of slowdown is tolerable. In Section 7.2, we will discuss several possible ways to further accelerate offline taint analysis.

Then we quantify the effects of the set of optimizations we adopted for the online instrumentation. Figure 4.13 shows the impact of instrumentation optimizations when applied cumulatively on SPEC CPU2006 with the *reference* workload. The "unopt" bar

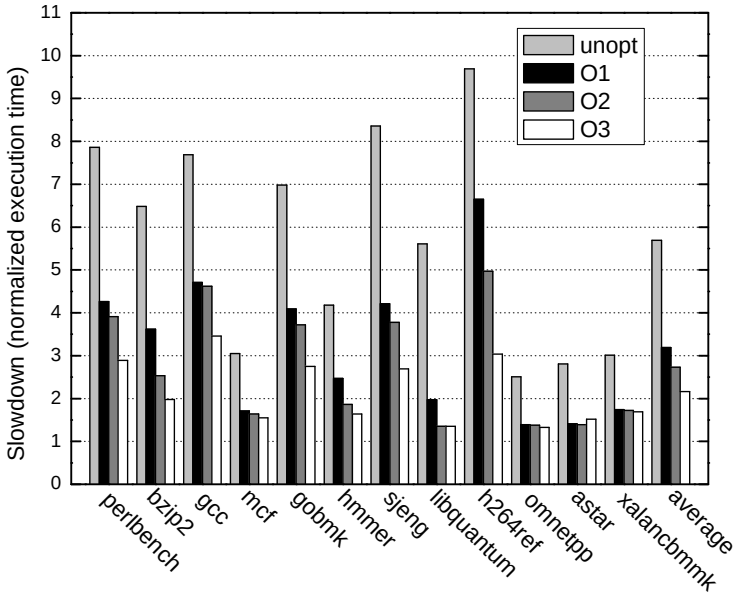

Figure 4.13. The impact of instrumentation optimization on SPEC CPU2006: O1 (inline analysis code), O2 (O1 + fast buffering APIs and call linkages), O3 (O2 + optimize rep-prefixed instructions).

approximates an un-optimized StraightTaint, which does not employ any optimization methods we discussed in Section 4.2 and Section 4.4. The bar denoted as O1, captures the effect of our effort on inlining analysis code, bringing a notable overhead step-down from average 5.69X to 3.19X. Optimization O2, which adds fast buffering APIs and call linkages, reduce running time further to 2.73X. Optimization to `REP`-prefixed instructions (O3) offers an enhanced performance improvement by average 26.4%, with a peak value 63.5% to the `h264ref`. After investigation, we find that as high as 13.4% of total executed instructions of the `h264ref` benchmark are `REP` instruction repetitions.

4.5.3 StraightTaint vs. FlowWalker

FlowWalker [27] is perhaps the closest work to StraightTaint in its goals: we are both offline taint analysis in *record and replay* style. Similar to StraightTaint, FlowWalker

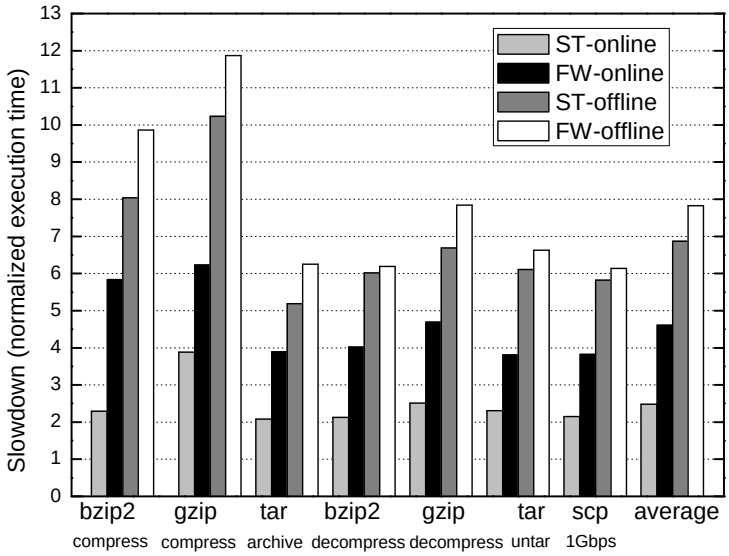

Figure 4.14. StraightTaint vs. FlowWalker: slowdown on common Linux utilities (Y axes shows the normalized execution time).

also records limited CPU context on top of Pin to calculate the memory address offline. However, FlowWalker lacks fine-grained optimizations in both online logging and offline taint analysis (see Section 3.4). In this experiment, we evaluate StraightTaint (short for ST) and FlowWalker (short for FW) on four common Linux utilities that represent three kinds of workload[2]. `tar` is I/O bounded, whereas `bzip2` and `gzip` are CPU intensive program. `scp` represents a middle level between these two cases. We use `tar` to archive and extract GNU Core utilities 8.13 package (∼50MB). And then we apply `bzip2` and `gzip` to compress and decompress the archive file of Core utilities. For `scp`, we copy the archive file of Core utilities over 1Gbps link. We achieve a similar improvement with the SPEC CPU2006 experiment. As shown in Figure 4.14, StraightTaint imposes a average 2.48X slowdown to native execution, with a 1.86 times speed up to FlowWalker. Besides, StraightTaint's offline taint analysis is faster than FlowWalker with a factor of 1.14. We attribute this to our sub-trace cache and function summary optimizations.

[2]SPEC2006's reference workload is too huge for FlowWalker to work out the result in reasonable time.

4.5.4 Offline Symbolic Taint Analysis

Table 4.1. StraightTaint successfully detected various intrusions with the listed exploits.

Program	Vulnerability	CVE ID	# Taint (Symbolic) Bytes		
			libdft	StraightTaint	pure SE
nginx	validation bypass	CVE-2013-4547	45	45	1,035
mini_httpd	validation bypass	CVE-2009-4490	66	66	2,706
libpng	denial of service	CVE-2014-0333	72	80	2,256
gzip	integer underflow	CVE-2010-0001	94	94	6,490
tiny server	validation bypass	CVE-2012-1783	125	131	12,171
coreutils	buffer overflow	CVE-2013-0221	252	272	–
libtiff	buffer overflow	CVE-2013-4231	268	280	–
waveSurfer	buffer overflow	CVE-2012-6303	384	384	–
grep	integer overflow	CVE-2012-5667	608	644	–
regcomp	validation bypass	CVE-2010-4052	1,124	1,186	–

Next we evaluate the accuracy of our offline symbolic taint analysis in the task of software attack detection. To this end, we test ten recent software vulnerabilities using a set of exploits listed in Table 4.1. These test cases are chosen from CVE vulnerability data source[3] with two criteria: 1) It is easy to mark the locations of taint sinks in the binary code so that we can count the tainted bytes at the same place; 2) we have exploits that can trigger these vulnerabilities (not all the CVE vulnerabilities have related exploits). All of these applications are compiled with the option "gcc -O2". Taking these exploits as inputs, we apply StraightTaint on each application and check taint tags at various taint sinks (e.g., function return value). In all cases, StraightTaint successfully detects the attacks without false negatives. At the same time, we count the number of tainted (or symbolic) bytes at the end of taint analysis. We compared StraightTaint with *Log-all* and *Pure SE*. Log-all means recording complete runtime data (e.g., each memory address and control transfer target) during online logging, and then use the data for offline taint analysis. Log-all represents vanilla decoupled offline taint analysis, but its

[3] `http://www.cvedetails.com/`

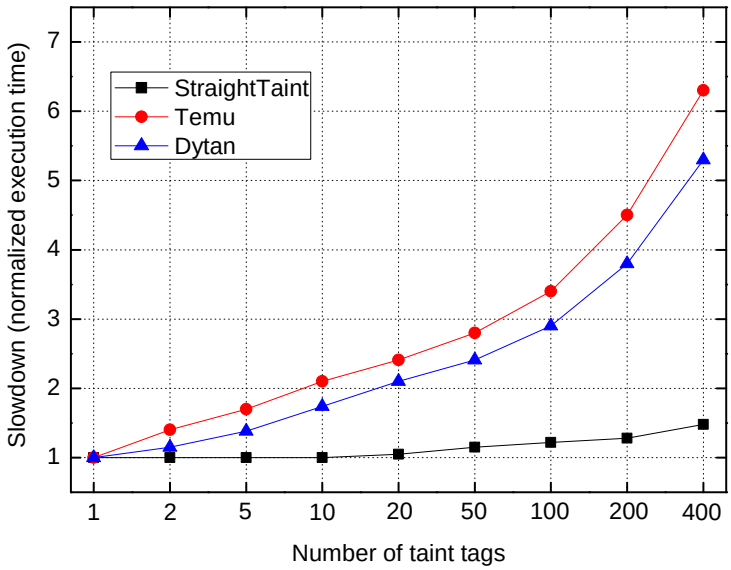

Figure 4.15. Normalized slowdown when the number of taint tags varies.

result is accurate. Pure SE does symbolic taint analysis but without concrete execution state initialization (see Section 4.3.2) and memory reference address resolution (see Section 4.3.3). As shown in Table 4.1, the taint bytes added by StraightTaint is quite close to the Log-all. StraightTaint introduces additional taint bytes to 6 cases, but no one is beyond 5%. Most likely, our conservative approach to dealing with symbolic memory indices results in the small additional taint bytes. In contrast, symbolic taint analysis with a completely unconstrained initial state (pure SE) incurs taint variable explosion. Pure SE fails in the last 5 test cases due to quickly reaching the memory capacity. Note that we also identify 14 code segments which can fail DTA tools with incomplete taint propagation strategies. One such example has been shown in Figure 4.8. In contrast, StraightTaint's full-featured offline taint analysis succeeds in all cases.

At last, we show that StraightTaint can support multi-tag taint analysis naturally. We test a lightweight web server, `thttpd`[4], with a 400-byte size HTTP request as

[4] `http://acme.com/software/thttpd/`

input. In our evaluation, 2 taint tags means that the first 200 bytes are labeled as one taint tag and the next 200 bytes are labeled as another taint tag; 400 taint tags means each input byte is associated with a different taint tag. Following the similar style, we vary the number of taint tags in each round. At the same time, we compare two DTA tools (Temu [64] and Dytan [15]) which can also perform multi-tag taint analysis, and Figure 4.15 shows the results. The baseline for each tool is their single-tag version. It is apparent that as the number of taint tags increases, both Temu and Dytan imposes high additional overhead, while StraightTaint only introduce 1.48X showdown in the worst case. Please note that this evaluation demonstrates StraightTaint's another notable feature: *once a log is captured, it can be analyzed multiple times*. In our multiple round testing, StraightTaint only needs to record the required data once and performs the different multi-tag propagation rounds on top on the straight-line code. By contrast, both Temu and Dytan have to rerun at each round.

4.6 Case Study: Attack Provenance Analysis

Because of the offline analysis property, StraightTaint is an ideal fit for *ex post facto* security applications. In this section, we demonstrate the merit of StraightTaint with a case study of attack provenance investigation. The goal is to reveal the provenance of intrusions or suspicious events (e.g., information leaks). The previous work [34, 87] did this by generating causal graph linking root causes and suspicious events. Certainly DTA can be utilized to precisely generate causal dependence between taint source and taint sink. We show that StraightTaint is able to get a similar level of precision as DTA with multi-tag backward propagation. The test case is `wget`[5], an open source tool for retrieving files from web. We execute `wget` with the command "`wget www.google.com`

[5] `http://www.gnu.org/software/wget`

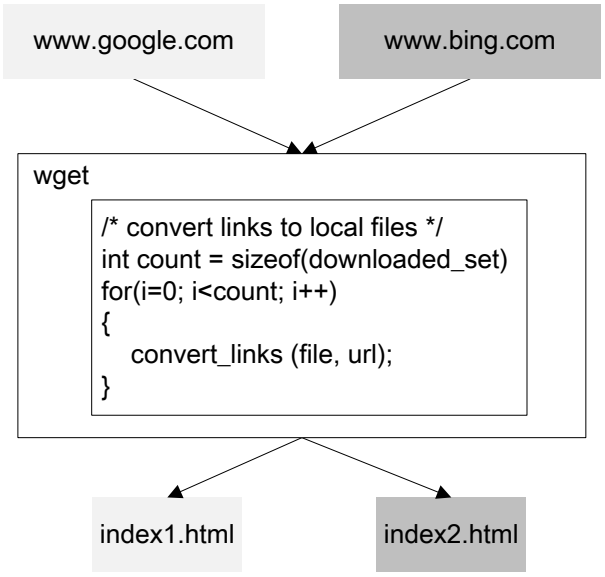

Figure 4.16. Causal relationship between two sinks and two sources.

`www.bing.com`".

As shown in Figure 4.16, `wget` receives two URLs as command line arguments and the downloads their respective `index.html` files (`index1.html` is from `www.google.com` and `index2.html` is from `www.bing.com`). Supposing we have already got these two downloaded files, an interesting question is "which exact URL are they derived from?" "google, bing or both?" Apparently DTA can precisely identify such mappings by forward taint tracking with multiple tags. Please note the pseudo-code of Figure 4.16: two files are generated subsequently when the loop is unrolling. As a result, static taint analysis, without runtime information, fails to identify causal relations between sources and sinks.

We take the input buffer of `fwrite`, which is used to generate HTML file, as symbolic taint sinks. Then we apply StraightTaint for backward tainting along the straight-line trace. Of course without runtime values and inputs, StraightTaint is unable to exactly correlates the concrete URL to its corresponding file. However, compared to pure static approaches, StraightTaint catches *conditional* causal relationships between

two sinks and sources: the first downloaded file is derived from the first URL input and the second one is related to the second URL. Another benefit of StraightTaint's *conditional tainting* is that we are possible to directly map previous taint analysis results to new inputs and runtime values. For example, supposing new command for *wget* is "`wget www.bing.com www.google.com`", with the previous conditional causal relationship, we can get the exact mappings immediately without running DTA again.

Chapter 5

TaintPipe: Pipelined Symbolic Taint Analysis

In this chapter, we exploit another style of parallelism, namely pipelining, to significantly improve the performance of dynamic taint analysis on top of the pervasive multi-core architectures. In our system [63], the execution of expensive taint analysis is decoupled from application execution to multiple pipeline stages running in parallel. Each worker thread can start running symbolic taint analysis very early even without knowing the explicit information, and will switch to the concrete taint analysis when the explicit information is ready. TaintPipe has two concurrently running parts: 1) the instrumented application thread doing a very lightweight online logging and acting as the source of the pipeline; 2) multiple worker threads as different stages of the pipeline to perform concrete/symbolic taint analysis in parallel. Each worker thread starts running symbolic taint analysis very early even without knowing the explicit taint tags and will switch to the concrete taint analysis when the explicit taint state is ready. In this way, the execution time of each worker thread may have many overlaps with other worker threads. With these overlaps, TaintPipe can make up the accumulated delay.

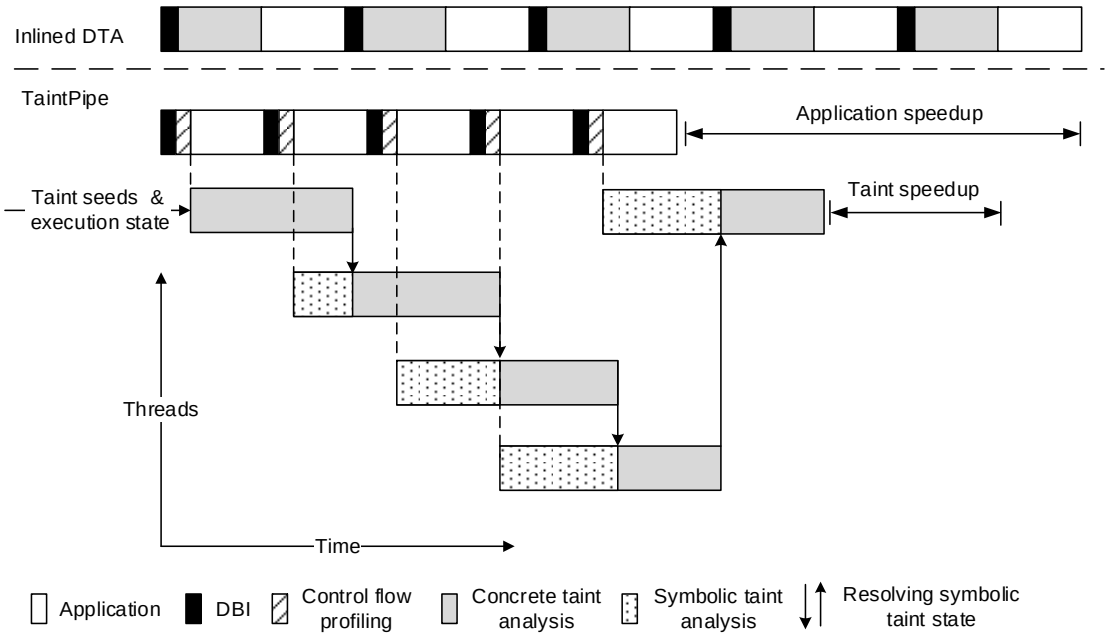

Figure 5.1. Inlined dynamic taint analysis vs. TaintPipe.

5.1 Background

In this section, we discuss the background and context information of the problem that TaintPipe seeks to solve. We start by comparing TaintPipe with the conventional inlined taint analysis approaches, and we then present the differences between the previous "primary & secondary" taint decoupling model and the pipelined decoupling style in TaintPipe.

5.1.1 Inlined Analysis vs. TaintPipe

Figure 5.1 ("Inlined DTA") illustrates a typical dynamic taint analysis mechanism based on dynamic binary instrumentation (DBI), in which the original program code and taint tracking logic code are tightly coupled. Especially, when dynamic taint analysis runs on the same core, they compete for the CPU cycles, registers, and cache space, leading to

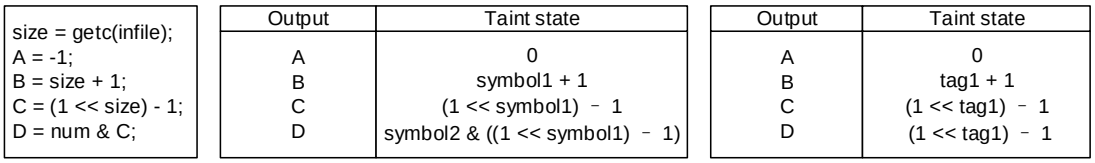

Figure 5.2. An example of symbolic taint analysis on a code segment: (a) code segment; (b) symbolic taint states, the input value `size` and `num` are labeled as `symbol1` and `symbol2`, respectively; (c) resolving symbolic taint states when `size` is tainted as `tag1` and `num` is a constant value (num = 0xffffffff).

significant performance slowdown. For example, "context switch" happens frequently between the original program instructions and taint tracking instructions due to the starvation of CPU registers. This means there will be a couple of instructions, mostly inserted per program instruction, to save and restore those register values to and from memory. At the same time, taint tracking instructions themselves (e.g., shadow memory mapping) are already complicated enough. One taint shadow memory lookup operation normally needs 6–8 extra instructions [21].

Our approach, analogous to the hardware pipelining, decouples taint logic code to multiple spare cores. Figure 5.1 ("TaintPipe") depicts TaintPipe's framework, which consists of two concurrently running parts: 1) the instrumented application thread performing lightweight online logging and acting as the source of the pipeline; 2) multiple worker threads as different stages of the pipeline to perform symbolic taint analysis. Each horizontal bar with gray color indicates a working thread. We start online logging when the predefined taint seeds are introduced to the application. The collected profile is passed to a worker thread. Each worker thread constructs a straight-line code segment and then performs taint analysis in parallel. In principle, fully parallelizing dynamic taint analysis is challenging because there are strong serial data dependencies between the taint logic code and application code [23]. To address this problem, we propose *segmented symbolic*

taint analysis inside each worker thread whenever the explicit taint information is not available, in which the taint state is symbolically calculated. The symbolic taint state will be updated later when the concrete data arrive. In addition to the control flow profile, the explicit execution state when the taint seeds are introduced is recorded as well. The purpose is to reduce the number of fresh symbolic taint variables.

We use a motivating example to introduce the idea of segmented symbolic taint analysis. Figure 5.2 shows an example for symbolic taint analysis on a *straight-line code segment*, which is a simplified code snippet of the `libtiff` buffer overflow vulnerability (CVE-2013-4231). Assume when a worker thread starts taint analysis on this code segment (Figure 5.2(a)), no taint state for the input data ("size" and "num" in our case) is defined. Instead of waiting for the explicit information, we treat the unknown values as taint symbols (`symbol1` for "size" and `symbol2` for "num", respectively) and summarize the net effect of taint propagation in the segment. The symbolic taint states are shown in Figure 5.2(b). When the explicit taint states are available, we resolve the symbolic taint states by replacing the taint symbols with their real taint tags or concrete values (Figure 5.2(c)). After that, we continue to perform concrete taint analysis like conventional DTA. Note that here we show pseudo-code for ease of understanding, while TaintPipe works on binary code.

Compared with inlined DTA, the application thread under TaintPipe is mainly instrumented with control flow profile logging code, which is quite lightweight. Therefore, TaintPipe results in much lower application runtime overhead. On the other hand, the execution of taint logic code is decoupled to multiple pipeline stages running in parallel. The accumulated effect of TaintPipe's pipeline leads to a substantial speedup on taint analysis.

5.1.2 "Primary & Secondary" Model

Some recent work [25, 26, 61] offloads taint logic code from the application (primary) thread to another shadow (secondary) thread and runs them on separate cores. At the same time, the primary thread communicates with the secondary thread to convey the necessary information (e.g., the addresses of memory operations and control transfer targets) for performing taint analysis. However, this model suffers from frequent communication between the primary and secondary thread. In principle, every memory address that is loaded or stored has to be logged and transferred. Due to the frequent synchronization with the primary thread and the extra instructions to access shadow memory, taint logic execution in the secondary thread is typically slower than the application execution. As a result, the delay for each taint operation could be accumulated, leading to an delay proportional to the original execution. ShadowReplica [26] partially addresses this drawback by performing advanced offline static optimizations on the taint logic code to reduce the runtime overhead. However, in many security analysis scenarios, precise static analysis and optimizations over taint logic code are not feasible, e.g., reverse engineering and malware forensics. In such cases, program static features such as control flow graphs are possibly obfuscated.

In TaintPipe, we record compact control flow information to reconstruct *straight-line* code, in which all the targets of direct and indirect jumps have been resolved. However, we do not record or transfer the addresses of memory operations. Our key observation is that most addresses of memory operations can be inferred from the straight-line code. For example, if a basic block is ended with an indirect jump instruction `jmp eax`, we can quickly know the value of `eax` from the straight-line code. In this way, all the other memory indirect access calculated through `eax` (before it is updated) can be determined. For instance, we can infer the memory load address for the instruction: `mov ebx,`

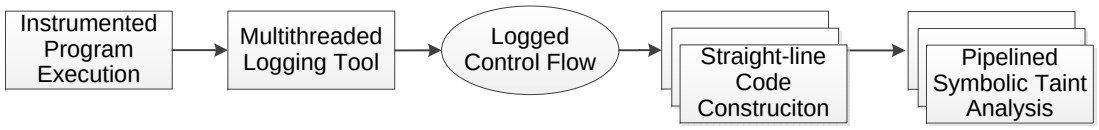

Figure 5.3. Architecture.

`[4*eax + 16]`. Even when the index of a memory lookup is a symbol, with the taint states and path predicates of the straight-line code, we can often narrow down the symbolic memory addresses to a small range in most cases.

Since TaintPipe's data communication is lightweight, TaintPipe can achieve nearly constant delay given enough number of worker threads. The upper limit number of worker threads is also bounded, which equals roughly the ratio of the taint analysis execution time over the application thread execution time for each segment.

Due to TaintPipe's pipelining design, it is possible that TaintPipe may detect an attack some time after the real attack has happened. However, this trade-off does not prevent TaintPipe from practically supporting a broad variety of security applications, such as attack forensic investigation and post-fact intrusion detection, which do not require strict runtime security enforcement. It is worth noting that different from ShadowReplica, TaintPipe does not depend on extensive static analysis to reduce data communication. Therefore, TaintPipe has a wider range of applications in speeding up analyzing obfuscated binaries, as static analysis of obfuscated binaries is of great challenge.

5.2 Design

Figure 5.3 illustrates the architecture of TaintPipe. We have built the pipelining framework on top of a dynamic binary instrumentation tool, enabling TaintPipe to work with unmodified program binaries. The steps followed by TaintPipe are as follows.

1. TaintPipe takes in a binary along with the taint seeds as input. The instrumented application thread starts execution with lightweight online logging for control flow and other information (Section 5.4.1).

2. Then the instrumented program is executed together with a multithreaded logging tool to efficiently deliver the logged data to memory (Section 5.4.2).

3. When the profile buffer becomes full, a taint analysis engine will be invoked for online pipelined taint analysis (Section 5.5.1).

4. The generated log data are then used to construct straight-line code, which helps to solve many precision loss problems in static taint analysis. In this stage, we generate a segment of executed code blocks for each logged data buffer. The memory addresses that are accessed through indirect jump targets are also resolved (Section 5.5.2).

5. The taint analysis engine will further translate straight-line code to taint operations, which avoid precision loss and support both multi-tag and bit-level taint analysis (Section 5.5.3).

6. With the constructed taint operations, TaintPipe performs pipelined symbolic taint analysis. When a thread finishes taint analysis with an explicit taint state, it synchronizes with its following thread to resolve the symbolic taint state (Section 5.5.4).

5.2.1 Segmented Symbolic Taint Analysis

In this section, we analyze symbolic taint analysis from a theoretical point of view to justify the correctness of our pipelining scheme. In order to formalize *segmented symbolic taint analysis*, we use the following notations:

1. Let σ denote a taint state, which maps variables to their taint tags.

2. Let $\mathcal{A}(\sigma, S)$ denote a symbolic taint analysis $\mathcal{A}$ on a straight-line code segment S, with an initial taint state σ. We use $\mathcal{A}_\sigma(S)$ for convenience.

Note that the straight-line code segment S has no control transfer statement. Conceptually, S only contains one type of statements, namely assignment statements. Of course, from the implementation point of view, there may be other types of statements, but they can all be regarded as assignment statements. For example, as we will show in Section 5.5.3, our taint operations contain assignment operations, laundering operations, and arithmetic operations. The latter two operations can be derived from taint assignment operations.

Based on the semantics of assignment statements, we define symbolic taint analysis for an assignment statement as follows:

$$\mathcal{A}_\sigma(x := e) = \sigma[x \mapsto e^t] \tag{5.1}$$

where e^t denotes the taint tag of e, and $[\cdot]$ is the taint state update operator. If x is a new variable, the taint state σ is extended with a new mapping from x to its taint. If x occurs in the taint state σ, for the variables in the domain of σ whose symbolic taint expressions depend on x, their symbolic taint expressions will be updated or recomputed with the new taint value of x.

Assume $\sigma_1 = \mathcal{A}_\sigma(i_1)$ for a statement i, then the symbolic taint analysis for two sequential statements $i_1; i_2$ is:

$$\mathcal{A}_\sigma(i_1; i_2) = \mathcal{A}_{\sigma_1}(i_2) \tag{5.2}$$

Assume straight-line code segment $S_1 = (i_1; S_1')$. We can then deduce the symbolic taint analysis on two sequential segments $S_1; S_2$ as follows:

$$\begin{aligned} & \mathcal{A}_\sigma(S_1; S_2) \\ =& \mathcal{A}_\sigma((i_1; S_1'); S_2) \\ =& \mathcal{A}_\sigma(i_1; (S_1'; S_2)) \\ =& \cdots \\ =& \mathcal{A}_{\mathcal{A}_\sigma(S_1)}(S_2) \end{aligned} \tag{5.3}$$

That is, given $\mathcal{A}_\sigma(S_1) = \sigma_1$ and $\mathcal{A}_\epsilon(S_2) = \sigma_2$, where ϵ is an empty taint state, Eq. 5.3 leads to:

$$\mathcal{A}_\sigma(S_1; S_2) = \sigma_2[\sigma_1] \tag{5.4}$$

Here, we misuse the taint state update operator $[\cdot]$ and apply it to a taint state map, instead of a single taint variable update. With Eq. 5.4, we can perform segmented taint analysis in parallel or in a pipeline style. For two segments $S_1; S_2$, assume the starting taint state is σ_0. We start two threads, one compute $\mathcal{A}_{\sigma_0}(S_1)$ and the other computes $\mathcal{A}_\epsilon(S_2)$, where ϵ is an empty taint state. Assume the result of the first thread analysis is σ_1 and the result of the second is σ_2. The symbolic taint analysis of $S_1; S_2$ is $\sigma_2[\sigma_1]$, that is, the right hand side of Eq. 5.4. Eq. 5.4 forms the foundation of our segmented taint analysis in a pipeline style.

5.3 Implementation

To demonstrate the efficacy of TaintPipe, we have developed a prototype on top of the dynamic binary instrumentation framework Pin [17] (version 2.12) and the binary analysis platform BAP [33] (version 0.8). The online logging and pipelining framework

are implemented as Pin tools, using about 3,100 lines of C/C++ code. The taint operation constructors are built on BAP IL (intermediate language). TaintPipe's taint analysis engine is based on BAP's symbolic execution module, using about 4,400 lines of Ocaml and running concurrently with Pin tools. We utilize Ocaml's functor polymorphism so that taint states can be instantiated in either concrete or symbolic style. All of the functionality implemented in taint analysis engine are wrapped as function calls. To support communication between Pin tools and taint analysis engine, we develop a lightweight RPC interface so that each worker thread can directly call Ocaml code. The saving and loading of the taint cache lookup table is implemented using the Ocaml Marshal API, which encodes IL expressions as sequences of compact bytes and then stores them in a disk file.

Dynamic binary instrumentation tools tend to inline compact and branch-less code to the final translated code. For the code with conditional branches, DBI emits a function call instead, which introduces additional overhead. Therefore, we carefully design our instrumentation code to favor DBI's code inlining. To fully reduce online logging overhead, we also utilize Pin-specific optimizations. We leverage Pin's fast buffering APIs for efficient data buffering. For example, the inlined `INS_InsertFillBuffer()` writes the control flow profile directly to the given buffer; the callback function registered in `PIN_DefineTraceBuffer()` processes the buffer when it becomes full or thread exits. Besides, we force Pin to use the fastcall x86 calling convention to avoid emitting stack-based parameter loading instructions (i.e., push and pop). Currently Pin-tools do not support the Pthreads library. Thus we employ Pin Thread API to spawn multiple worker threads. We also implement a counting semaphore based on Pin's locking primitives to assist thread synchronization.

5.4 Logging

TaintPipe's pipeline stages consist of multiple threads. The thread of instrumented application (producer) serves as the source of pipeline, and a number of Pin internal threads act as worker threads to perform symbolic taint analysis on the data collected from the application thread. Note that unlike application threads, worker threads are not JITed and therefore execute natively. One of the major drawbacks of previous dynamic taint analysis decoupling approaches is the large amount of information collected in the application thread and the high overhead of communication between the application thread and analysis thread. To address these challenges, TaintPipe performs lightweight online logging to record information required for pipelined taint analysis. The logged data comprise control flow profile and the concrete execution state when taint seeds are first introduced, which is the starting point of our pipelined taint analysis. The initial execution state, consisting of concrete context of registers and memory, (e.g., CR0$\sim$CR4, EFLAGS and addresses of initial taint seeds), is used to reduce the number of fresh symbolic taint variables.

We take major two steps to reduce the application thread slowdown: First, we adopt a compact profile structure so that the profile buffer contains logged data as much as possible, and it is quite simple to recover the entry address of each basic block as well. Second, we apply the "one producer, multiple consumers" model and N-way buffering scheme to process full buffers asynchronously, which allows application to continue execution while pipelined taint analysis works in parallel. We will discuss each step in the following sub-sections.

5.4.1 Lightweight Online Logging

Besides the initial execution state when the taint seeds are introduced, TaintPipe collects control flow information, which is represented as a sequence of basic blocks executed. Conceptually, we can use a single bit to record the direction of conditional jump [84], which leads to a much more compact profile. However, reconstruction straight-line code from 1 bit profile is more complicated to make it fit for offline analysis. Zhao et al. [82] proposed *Detailed Execution Profile* (DEP), a 2-byte profile structure to represent 4-byte basic block address on x86-32 machine. In DEP, a 4-byte address is divided into two parts: *H-tag* for the 2 high bytes and *L-tag* for 2 low bytes. If two successive basic blocks have the same H-tag, only L-tag of each basic block enters the profile buffer; otherwise a special tag 0x0000 followed by the new H-tag will be logged into the buffer.

We extend DEP's scheme to support `REP`-prefix instructions. A number of x86 instructions related to string operations (e.g., `MOVS`, `LODS`) with `REP`-prefix are executed repeatedly until the counter register (ecx) counts down to 0. Dynamic binary instrumentation tools [17,83] normally treat a `REP`-prefixed instruction as an implicit loop and generate a single instruction basic block in each iteration. In our evaluation, there are several cases that unrolling such `REP`-prefix instructions would be a performance bottleneck. We address this problem by adding additional escape tags to represent such implicit loops. Figure 5.4 presents an example of the control flow profile we adopted. The left part shows a segment of straight-line code containing 1028 basic blocks, and 1024 out of them are due to `REP`-prefixed instruction repetitions. Our profile (the right side of Figure 5.4) encodes such case with two consecutive escape tags (0xffff), followed by the number of iterations (0x0400).

We note that it is usually unnecessary to turn on the logging all the time. For example, when application starts executing, many functions are only used during loading. At that

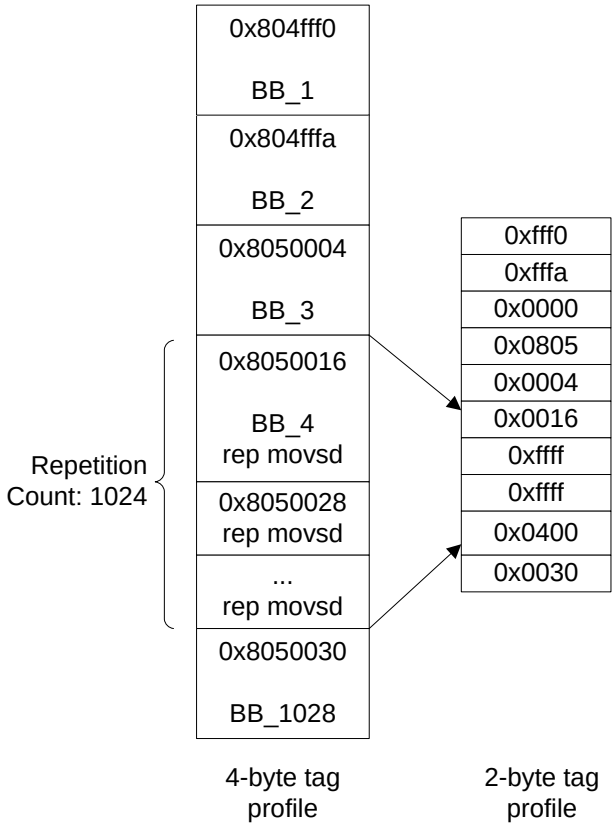

Figure 5.4. An example of 2-byte tag profile.

time, no sensitive taint seed is introduced. Therefore we perform on-demand logging to record control flow profile when necessary. As application starts running, we only instrument limited functions to inspect the various input channels that taint could be introduced into the application (taint seeds). Such taint seeds include standard input, files and network I/O. Besides, users can customize other values as taint seeds, such as function return values or parameters. When the pre-defined taint seeds are triggered, we turn on the control flow profile logging. At the same time, we save the current execution state to be used in the pipelined taint analysis. Many well-known library functions have explicit semantics, which facilitates us to selectively turn off logging inside these functions and propagate taint correspondingly at function level. We will discuss this issue further in Section 5.5.3.

5.4.2 N-way Buffering Scheme

Since TaintPipe's online logging is lightweight, application (producer) thread's execution speed is typically faster than the processing speed of worker threads. To mitigate this bottleneck, we employed "one producer, multiple consumers" model and *N-way buffering scheme* [88]. At the center of our design is a thread pool, which is subdivided into n linked buffers, and the producer thread and multiple worker threads work on different buffers simultaneously. More specifically, when the instrumented application thread starts running, we first allocate n linked empty buffers ($n > 1$). At the same time, n Pin internal threads (worker threads) are spawned. Each worker thread is bound to one buffer and communicates with the application thread via semaphores. When a buffer becomes full, the application thread will release the full buffer to its corresponding worker thread and then continue to fill in the next available empty buffer. Given a full profile buffers, a worker thread will send it to a taint analysis engine to perform concrete/symbolic taint analysis in parallel. After that, the worker thread will release the profile buffer back to the application thread and wait for processing the next full buffer.

It is apparent that the availability of unused worker threads and the size of profile buffer will affect overall performance of TaintPipe (both application execution time and pipelined taint analysis) significantly. In Section 5.6.1, we will conduct a series of experiments to find the optimal values for these two factors.

5.5 Symbolic Taint Analysis

When the application thread releases a full profile buffer, a worker thread is waked up to capture the profile buffer and then communicates with a taint analysis engine for symbolic taint analysis.

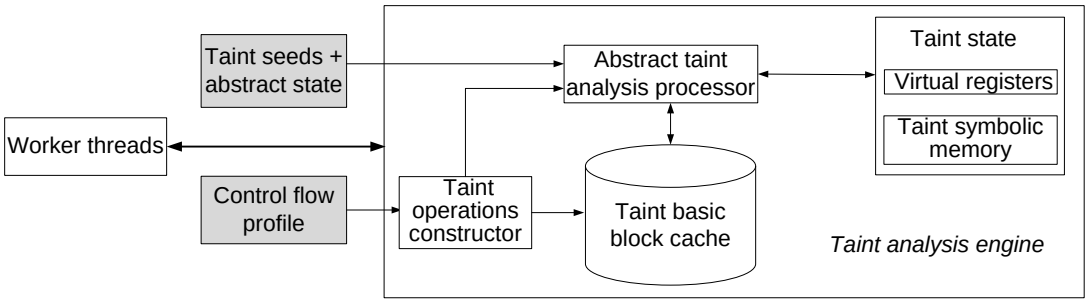

Figure 5.5. The structure of taint analysis engine.

5.5.1 Taint Analysis Engine

The taint analysis engine will first convert the control flow profile to a segment of straight-line intermediate language (IL) code and then translates the IL code to even simpler taint logic operations. The translations are cached for efficiency at taint basic block level. The key components of taint analysis engine are illustrated in Figure 5.5.

The core of TaintPipe's taint analysis engine is an abstract taint analysis processor, which simulates a segment of taint operations and updates the taint states accordingly. The taint state structure contains two contexts: virtual registers keeping track of symbolic taint tags for register, and taint symbolic memory for symbolic taint tags in memory. The taint symbolic memory design is like the two-level page table structure and each page of memory consists of symbolic taint formulas rather than concrete values. After the initialization of the symbolic taint inputs, the engines perform taint analysis either concretely or symbolically in a pipeline style.

5.5.2 Straight-line Code Construction

Given the control flow profiles, recovering each basic block's H-tag and L-tag is quite straightforward. A basic block's entry address is the concatenation of its corresponding H-tag and L-tag [82]. The taint analysis engine should only execute the instructions

required for taint propagation. Otherwise, the work thread may run much slower than the application thread. On the other side, due to the cumbersome x86 ISA, precisely propagating taint for the complex x86 instructions is an arduous work, especially for some instructions with side effect of conditional taint (e.g., `CMOV`). To achieve these two goals, we first extract the x86 instructions sequence from the application binary and then lift them to BIL [33], a RISC-like intermediate language. Since we know exactly the execution sequence, the sequence is a straight-line code. We have removed all the direct and indirect control transfer instructions and substituted them with control transfer target assertion statements.

After resolving an indirect control transfer, we go one step further to determine all the memory operation addresses which depend on this indirect control transfer target. For example, after we know the target of `jmp eax`, we continue to trace the use-def chain of `eax` for each memory load or store operation whose address is calculated through this `eax`. With the initial execution state (containing addresses of taint seeds) and indirect control transfer target resolving, we are able to decide most of the memory operation addresses.

For some applications such as word processing, a symbolic taint input may be used as a memory lookup index. Without any constraint, a symbolic memory index could point to any memory cell. Inspired by the index-based memory model proposed by Cha et al. [89], we attempt to narrow down the symbolic memory accesses to a small range with symbolic taint states and path predicates. We first leverage value set analysis [90] to limit the range of a symbolic memory access and then refine the range by querying a constraint solver. The path predicate along the straight-line code usually limits the scope of symbolic memory access. Figure 5.6 shows such an example where the path predicate restricts the symbolic memory index i within a range such that $7 < i < 10$.

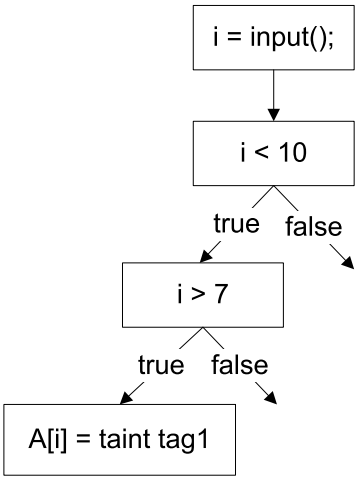

Figure 5.6. A path predicate constrains symbolic memory access within the boundary of $7 < i < 10$.

When propagating a taint tag to the memory cell referenced by i, we conservatively taint all the possible memory slots, that is, `A[8]` and `A[9]` in Figure 5.6 will be tainted as `tag1`. In Section 5.7, we will demonstrate that our symbolic memory index solution only introduces marginal side effects.

5.5.3 Taint Operation Generation

Based on BIL statements, we construct taint operations. Taint operations inside a basic block are formed as "taint basic block" [28], which are cached for efficiency. To make the best of cache effect, we merge the basic blocks with only one predecessor and one successor. Since BIL explicitly reveals the side effect of intricate x86 instructions, it is easy to perform intra-block optimizations to get rid of redundant taint operations. Therefore, our taint operations are simple and accurate. Currently our taint operations mainly consist of three types of operations for tracking taint data:

1. Assignment operations: The operations in this category are involved in copying

values between registers and registers/memory. We simply assign the taint tag of source operand to the destination operand.

2. Laundering operations: The operations are used to clean the taint tag of the destination operand. For example, `xor eax,eax` will clean the taint result of `eax`. We identify all laundering operations in taint basic blocks and substitute them with assignment operations.

3. Arithmetic and logic operations: This category of operations are the most difficult to handle. We emulate arithmetic and logic operations on the taint tags to capture their real semantics.

Figure 5.7 presents an example of taint operations for a basic block. TaintPipe's symbolic taint operations outperform conventional DTA approaches in three ways. First and foremost, multi-tag taint analysis is straightforward for TaintPipe. Each symbolic variable can naturally represent a taint tag (see Line 1 and Line 2 in Figure 5.7). Second, previous DTA tools mostly adopted a "short circuiting" method to handle arithmetic operations, that is, the destination is tainted if at least one of the source operands is tainted regardless of the real semantics. However, in many scenarios, it will lead to precision loss. Check the code at Line 4 and 5 in Figure 5.7, value d will always be zero since b is the negation of a. Unfortunately, some previous work may label d as tainted incorrectly [31]. Third, different from related work [19, 26], TaintPipe supports bit-level taint for EFLAGS register, representing whether a bit of the EFLAGS is tainted or not due to side effects. Recent work has demonstrated the value of bit-level taint in binary code de-obfuscation [38].

Another major optimization we adopt is so called "function summary". As many well-known library functions have explicit semantics (e.g., atoi, strlen), we generate a summary of each function and propagate taint correspondingly at function level. Table 5.1

```
int a, b, c, d;
1: a = read ();          1: Taint (a) = tag1;
2: c = read ();          2: Taint (c) = tag2;
3: c = c xor c;          3: Taint (c) = 0;
4: b = ~ a;              4: Taint (b) = ~ tag1;
5: d = a & b;            5: Taint (d) = 0;
```

(a) a basic block (b) a taint basic block

Figure 5.7. Example: taint operations.

Table 5.1. Function summary.

Category	Function
No tainting	strcmp, strncmp, memcmp, strlen, strchr, strstr, strpbrk, strcspn, qsort, rand, time, clock, ctime
Function level	strcat, strncat, strcpy, strncpy, memcpy, memmove, strtok, atoi, itoa, abs tolower, toupper

lists two types of function summary TaintPipe supports: 1) Functions within "no tainting" category do not have any side effect on taint state. We can safely turn off logging when executing them. 2) Some functions do propagate taint from an input parameter to output. We still turn off logging and update taint state correspondingly when these functions return.

5.5.4 Symbolic Taint State Resolution

In TaintPipe's pipeline framework, a worker thread may perform taint analysis concretely or symbolically in parallel. When a worker thread completes taint analysis with concrete taint tags, the final taint state it maintains is deterministic. Then it synchronizes with the subsequent worker thread to resolve the symbolic taint state maintained by the latter. Taint states are allocated in a shared memory area so that multiple threads can access them easily. Basically, given the concrete taint state at the beginning of a code segment, we replace a symbolic taint tag with the appropriate starting value (either a taint tag or a

concrete value). We further update all the symbolic formula containing that symbolic taint tag. For example, the logic AND formula in Figure 5.2(b) will be simplified to a single taint tag. After that, the subsequent thread switches to the concrete taint analysis and continue processing the left segment code.

In this order, the taint states of each segment will be resolved and updated one by one. The defined taint policy (e.g., a function return value should not be tainted) is checked along the concrete taint analysis as well. A tainted sink is identified if it contains a symbolic formula; multiple tags are determined by counting the number of different symbols in the formula. Note that a previous hardware-assisted approach [23] utilized a separate "master" processor to update each segment's taint status sequentially. However, as pointed out by the paper, when there are more than a few "worker" processors, the master processor will become the bottleneck. Our approach amortizes the workload of the master processor to each worker thread.

5.6 Performance Evaluation

We conducted experiments with several goals in mind. First, we wanted to choose optimal values for two factors that may affect TaintPipe's performance, namely control flow profile buffer size and the number of worker threads. Then we studied overall runtime overhead when running TaintPipe on the SPEC2006 int benchmarks and a number of common utilities. We also compared TaintPipe with a highly optimized inline dynamic taint analysis tool. We wanted to make sure TaintPipe is effective in speeding up various security analysis tasks and can compete with conventional inlined dynamic taint analysis in precision. We demonstrated three compelling applications: 1) detecting software attacks; 2) tracking information flow in obfuscated malicious programs.

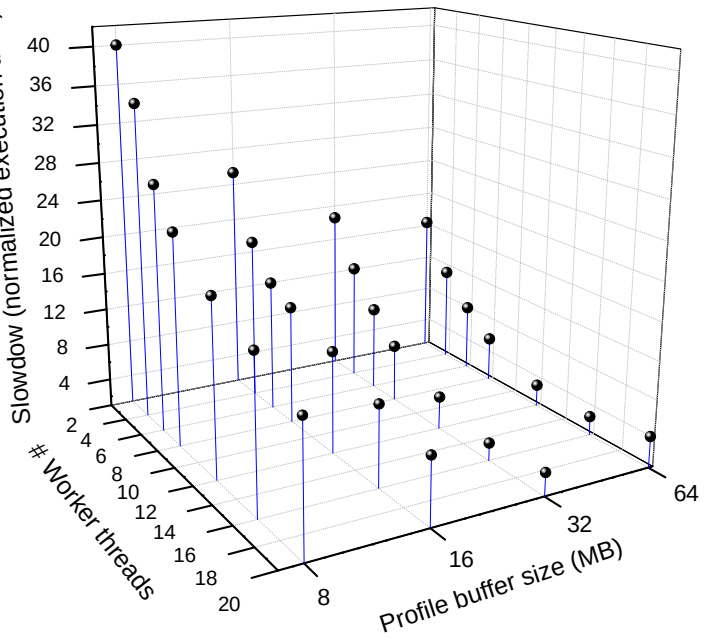

Figure 5.8. Optimal buffer size and number of worker threads.

5.6.1 Experiment Setup

Our experiment system has two Intel Xeon E5-2690 processors, 128GB of memory, and a 250GB solid state drive, running Ubuntu12.04. Each processor is equipped with 8 2.9GHz cores, 16 hyper threads, and 20MB L3 cache. The performance data reported in this section are all mean values with 5 repetitions.

TaintPipe's performance is affected by the size of the profile buffers and the number of worker threads. Generally, the more worker threads and larger profile buffers, the less possibility that an application is suspended to wait for a free buffer. On the other hand, our taint analysis engine has to take longer time to process larger segment code. We conducted a series of tests with the SPEC CPU2006 int benchmarks, under different settings of these two variables. We dynamically adjust the number of worker threads from 2 to 20 (2, 4, 6, 8, 12, 16 and 20), and profile size from 8MB to 64MB (8MB,

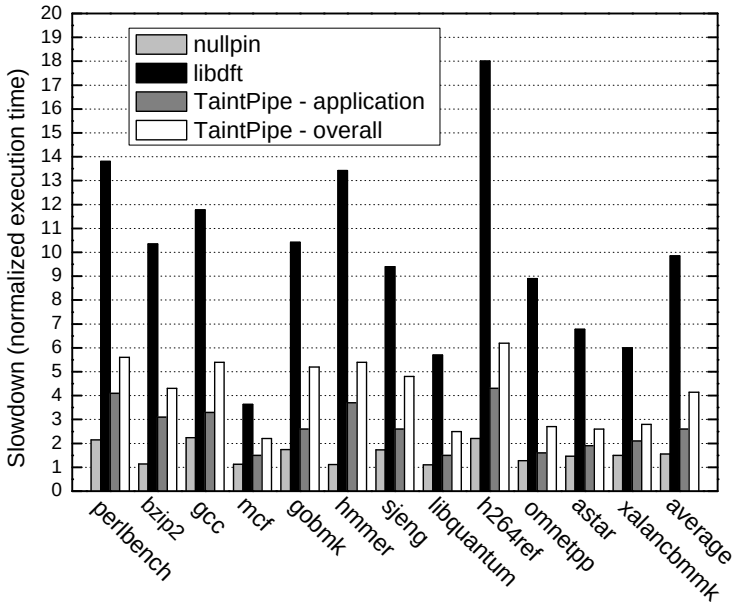

Figure 5.9. TaintPipe slowdown on SPEC CPU2006.

16MB, 32MB and 64MB). Figure 5.8 displays the experimental results. Roughly, as number of worker threads and buffer size increase, the application slowdown reduces. That is mainly because large buffer sizes allow application thread continue to fill up and worker threads spend less time on synchronization. After a certain point (buffer size $\geq$ 32MB and number of worker threads $>$ 16), overhead increases slightly. Two factors prevent TaintPipe from achieving more speedup. First, taint analysis engine slows down when processing large code segment. Second, more worker threads introduce larger communication latency when resolving symbolic taint states. According to the results, we set the two factors as their optimal values (32MB buffer size and 16 worker threads), which will be used in the following experiments.

To evaluate the performance gains achieved by pipelining taint logic, we compared TaintPipe with a state-of-the-art tool, libdft [19], which performs inlined dynamic taint analysis based on Pin ("libdft" bar). In addition, we developed a simple tool to measure the slowdown imposed exclusively by TaintPipe. It runs a program under Pin without any

form of analysis ("nullpin" bar). The "TaintPipe - application" bar represents the running time of instrumented application thread alone, and "TaintPipe - overall" corresponds to the overall overhead when both the application thread and pipelined worker threads are running. The major reason we reported "TaintPipe - application" and "TaintPipe - overall" time separately is to show the two improvements, namely "Application speedup" and "Taint speedup" (see Figure 5.1). Since the application thread typically runs faster than the worker threads, the "TaintPipe - overall" time is actually dominated by the worker threads. Therefore, usually the "TaintPipe - overall" time represents the relative time spent by worker threads as well. The times reported in this section are all normalized to native execution (the numbers of Y axes), that is, application running time without dynamic binary instrumentation.

5.6.2 SPEC CPU2006

Figure 5.9 shows the normalized execution times when running the SPEC CPU2006 int benchmark suite under TaintPipe. On average, the instrumented application thread enforces a 2.60X slowdown to native execution, while the overall slowdown of TaintPipe is 4.14X. If we take Pin's environment runtime overhead ("nullpin" bar) as the baseline, we can see TaintPipe imposes 2.67X slowdown ("TaintPipe - overall" / "nullpin") and libdft introduces 6.4X slowdown—this number is coincident to the observation that propagating a taint tag normally requires extra 6–8 instructions [20, 21]. In summary, TaintPipe outperforms inlined dynamic taint analysis drastically: 2.38X faster than the inlined dynamic taint analysis, and 3.79X faster in terms of application execution. In the best case (*h264ref*), the application speedup under TaintPipe exceeds 4.18X.

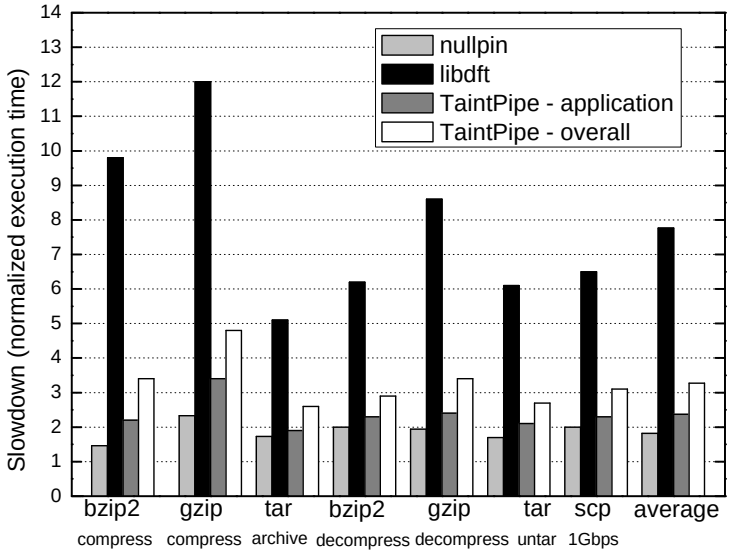

Figure 5.10. TaintPipe slowdown on common Linux utilities.

5.6.3 Utilities

We also evaluated TaintPipe on four common Linux utilities, which were not chosen randomly. These four utilities represent three kinds of workloads: I/O bounded (`tar`), CPU bounded (`bzip2` and `gzip`), and the case in-between (`scp`). We applied `tar` to archive and extract the GNU Core utilities package (version 8.13) (∼50MB), then we employed `bzip2` and `gzip` to compress and decompress the archive file. Finally we utilized `scp` to copy the archive file over a 1Gbps link. As shown in Figure 5.10, TaintPipe reduced slowdown of dynamic taint analysis from 7.88X to 3.24X, by a factor of 2.43 on average.

5.6.4 Apache and MySQL

In addition to SPEC CPU2006 benchmark and common utilities, we also evaluate TaintPipe with two larger and more complex software: Apache web server and MySQL

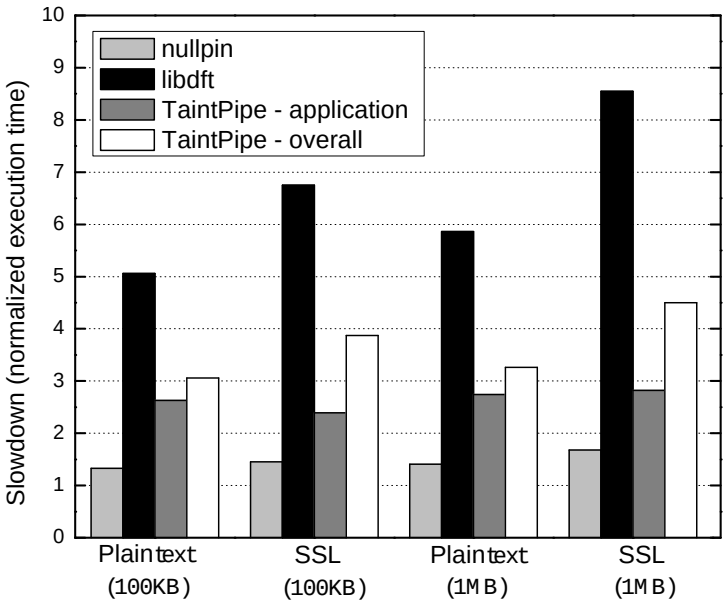

Figure 5.11. TaintPipe slowdown on Apache web server.

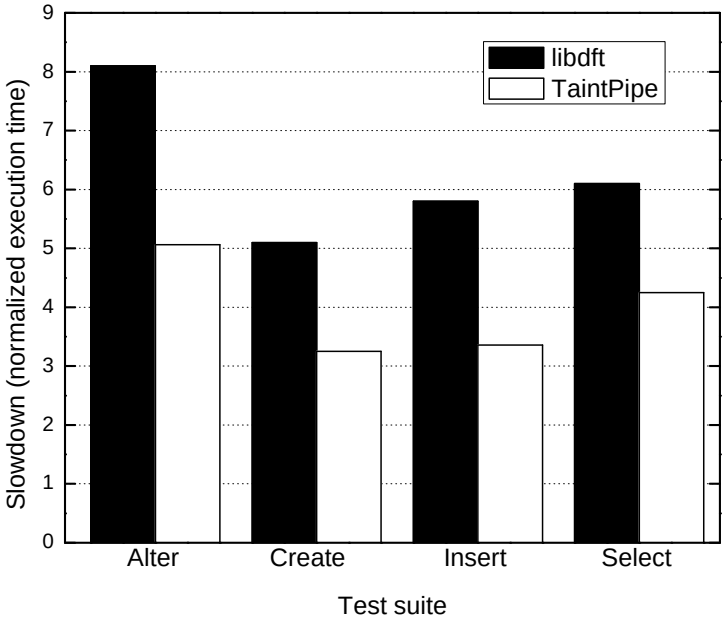

Figure 5.12. TaintPipe slowdown on MySQL DB server.

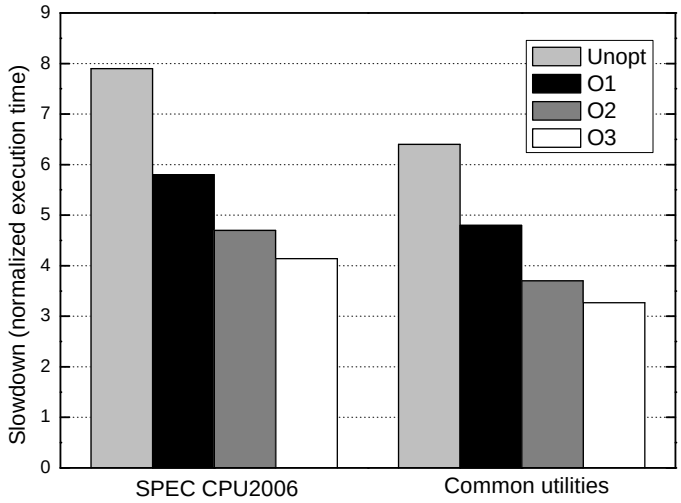

Figure 5.13. The impact of optimizations to speed up TaintPipe when applied cumulatively: O1 (function summary), O2 (O1 + taint basic block cache), O3 (O2 + intra-block optimizations).

DB server. We tested Apache v2.2.24 and set all settings as their default options. We executed Apache under libdft and TaintPipe, respectively, using Apache's utility `ab` and static HTML files with different file size (100KB and 1MB). Besides, we also ran Apache when the SSL/TLS encryption is enabled (i.e., SSL). Figure 5.11 shows the result. On average, TaintPipe introduces 3.2X slowdown (5.5X for libdft) when testing with plaintext. Since the intensive cryptographic operations performed by SSL are CPU bounded, the slowdown is larger when SSL is enabled (4.19X for TaintPipe and 7.75X for libdft). Figure 5.12 present the results from evaluating MySQL DB server. We tested MySQL v5.0.51b and its own benchmark suite (sql-bench). The benchmark suite has four different tests cases that measure the completion time of various DB operations such as table creation, table modification, data selection, and insertion. The average slowdown introduced by TaintPipe is 3.98X and 6.3X for libdft.

5.6.5 Effects of Optimizations

In this experiment, we quantify the effects of taint logic optimizations we presented in Section 5.5, which are paramount for optimized TaintPipe performance. Figure 5.13 shows the impact of these optimizations when applied cumulatively on SPEC CPU2006 and the set of common utilities. The "unopt" bar approximates an un-optimized TaintPipe, which does not adopt any optimization method. The "O1" bar indicates the optimization of function summary, reducing application slowdown notably by 26.6% for SPEC CPU2006 and 25.0% for the common utilities. The "O2" bar captures the effect of taint basic block cache, leading to a further reduction by 19.0% and 22.9% for SPEC and utilities, respectively. Intra-block optimizations, denoted by "O3", offer further improvement, 12.0% with SPEC and 11.6% with the utilities).

5.7 Security Applications

In the section, we demonstrate TaintPipe's appealing applications to software attack detection and malware taint graph generation.

5.7.1 Software Attack Detection

One important application of taint analysis is to define taint policies, and ensure they are not violated during taint propagation. We tested TaintPipe with 12 recent software exploits listed in Table 5.2, which covers a wide range of real-life software vulnerabilities. For example, the vulnerabilities in `nginx`, `micro_httpd`, and `tiny server` allow remote attackers to bypass input validation and crash the program. The `libtiff` buffer overflow vulnerability leads to an out of bounds loop limit via a malformed gif image. Both `boa` and `thttpd` write data to a log file without sanitizing non-printable

Table 5.2. Tested software vulnerabilities.

Program	Vulnerability	CVE ID	# Taint Bytes		
			libdft	Temu	TaintPipe
Nginx	Validation Bypass	CVE-2013-4547	45	45	45
Micro_httpd	Validation Bypass	CVE-2014-4927	80	85	80
Tiny Server	Validation Bypass	CVE-2012-1783	125	126	125
Regcomp	Validation Bypass	CVE-2010-4052	1,124	1,148	1,180
Libpng	Denial Of Service	CVE-2014-0333	72	72	72
Gzip	Integer Underflow	CVE-2010-0001	94	112	96
Grep	Integer Overflows	CVE-2012-5667	608	682	653
Coreutils	Buffer Overflow	CVE-2013-0221	252	260	256
Libtiff	Buffer Overflow	CVE-2013-4231	268	286	290
WaveSurfer	Buffer Overflow	CVE-2012-6303	384	418	406
Boa	Information Leak	CVE-2009-4496	164	164	164
Thttpd	Information Leak	CVE-2009-4491	328	328	328

characters, which may be exploited to execute arbitrary commands. Since we have detailed vulnerability reports, we can easily mark the locations of taint sinks in the straight-line code and set corresponding taint policies. For example, `gif2tiff.c` in `libtiff` takes an integer variable stemming from input as a loop limit (line 13 in Figure 5.14). However, this variable can be manipulated as a large number via a malformed gif image. An out-of-bound loop limit leads to memory corruption.

In our evaluation, TaintPipe did not generate any false positives and successfully identified taint policy violations while incurring only small overhead. At the same time, we evaluated the accuracy of TaintPipe. To this end, we counted the total number of tainted bytes in the taint state when taint analysis hit the taint sinks. Column $4 \sim 6$ of Table 5.2 show the number of taint bytes when running libdft, Temu [64] and TaintPipe, respectively. Compared with the inlined dynamic taint analysis tools (libdft and Temu), TaintPipe's symbolic taint analysis achieves almost the same results in 8 cases and introduces only a few additional taint bytes in the other 4 cases. We attribute this to our conservative approach to handling of symbolic memory indices. The evaluation data

```
1  unsigned int prefix[4096];
2  unsigned char suffix [4096];
3  int datasize, codesize, codemarsk; /* decoder working variables */
4  int clear, eoi, avail, oldcode; /* special code values */
5  …  //omitted
6  datasize = getc(infile);  /* infile is a picture with gif format */
7  clear = 1 << datasize;
8  eoi = clear + 1;
9  avail = clear + 2;
10 oldcode = -1;
11 codesize = datasize + 1;
12 codemask = (1 << codesize) - 1;
13 for (code = 0; code < clear; code++) {    /* access violation */
14 prefix[code] = 0;
15 suffix[code] = code;
16 }
17 …  //omitted
```

Figure 5.14. Libtiff vulnerability (CVE-2013-4231).

Table 5.3. Malware samples and taint graphs.

Sample	Type	Taint Graph		Control Flow
		Node #	Edge #	Obfuscation
Svat	Virus	90	62	
RST	Virus	154	82	
Agent	Rootkit	624	402	✓
KeyLogger	Trojan	554	368	✓
Subsevux	Backdoor	1648	764	✓
Tsunami	Backdoor	734	534	✓
Keitan	Backdoor	618	482	✓
Fireback	Backdoor	1038	620	✓

show that TaintPipe does not result in over-tainting [14] and rivals the inlined dynamic taint analysis at the same level of precision.

5.7.2 Malware Taint Graph Generation

We ran 8 malware samples collected from `VX Heavens`[1] with TaintPipe.[2] Similar to Panorama [10], we tracked information flow and generated a *taint graph* for each sample.

[1] http://vxheaven.org

[2] All these 8 samples are not packed. To analyze packed binaries, we can start TaintPipe when the unpacking procedure arrives at the original entry point.

In a taint graph, nodes represent taint seeds or instructions operating on taint data, and a directed edge indicates an explicit data flow dependency between two nodes. Taint graph faithfully describes intrinsic malicious intents, which can be used as malware specification to detect suspicious samples [91]. The statistics of our testing results are presented in Table 5.3. It is worth noting that 6 out of 8 malware samples are applied with various control flow obfuscation methods (the fifth column), such as opaque predicates, control flow flattening, obfuscated control transfer targets, and call stack tampering. As a result, the control flow graphs are heavily cluttered. For example, malware samples `Keitan` and `Fireback` have a relatively high ratio of indirect jumps (e.g., `jmp eax`). Typically it is hard to precisely infer the destination of an indirect jump statically. Thus, the taint logic optimization methods that rely on accurate control flow graph [26, 92] will fail. In contrast, our approach does not rely on control flow graph and therefore we analyzed these obfuscated malware samples smoothly.

Chapter 6

Applications to Reverse Engineering

In addition to runtime security policies enforcement, taint analysis has been widely applied in a variety of "post-fact" security analysis applications as well, such as attack provenance investigation [34], computer forensic analysis [36], malware analysis [9, 10], and reverse engineering [32, 37, 38]. Dynamic taint analysis in these applications still suffers from high runtime overhead. Therefore, out decoupled taint analysis approaches are particularly well suited for such post fact security analysis tasks. Our *full-featured* symbolic taint analysis can achieve better accuracy and performance when analyzing obfuscated code. In addition, our current version of symbolic taint analysis can be extended to support more fine-grained taint tracking. For example, we can associate each 4-byte application word with an 8-byte metadata in the shadow memory. When a 4-byte word is tainted, the first 4-byte of the metadata records current instruction address (i.e., the value of `eip`), and the left 4-byte stores the address of the source operand. In this way, we can reconstruct a detailed taint propagation trace. To make the techniques described in this thesis applicable to a broader range of such applications, we are interesting in

integrating the decoupled symbolic multi-tag taint analysis into the tasks of cryptography function detection and speeding up semantics-based binary diffing. Our technique can significantly speed up the tedious work in these applications.

Furthermore, we have designed a binary code control flow profiling tool, *BinCFP*, as a part of TaintPipe's pipeline infrastructure and StraightTaint's multithreaded fast buffering scheme. BinCFP allows efficient and complete binary code control flow profiling on ubiquitous multi-core platforms. In many tasks of reverse engineering and binary code analysis (e.g., hybrid disassembly, resolving indirect jump, and decoupled taint analysis), the knowledge of detailed dynamic control flow can be of great value. However, the high runtime overhead beset the complete collection of dynamic control flow. The previous efforts on efficient path profiling cannot be directly applied to the obfuscated binary code in which an accurate control flow graph is typically absent. To address these challenges, we develop BinCFP, an efficient multi-threaded binary code control flow profiling tool by taking advantage of pervasive multi-core platforms. BinCFP relies on dynamic binary instrumentation to work with the unmodified binary code. The key of BinCFP is a multi-threaded fast buffering scheme that supports processing trace buffers asynchronously. To achieve better performance gains, we also apply a set of optimizations to reduce control flow profile size and instrumentation overhead. Our design enables the complete dynamic control flow collection for an obfuscated binary execution. We will introduce our technique applications to reverse engineering tasks in the following sections.

6.1 Binary Code Control Flow Profiling

Dynamic control flow information is typically represented as a sequence of basic blocks executed during run time [82]. In many applications of reverse engineering and binary code analysis, the detailed dynamic control flow can be very handy. For example,

control flow information can increase the accuracy of static disassembly [93] and reverse the effect of control flow obfuscation [94] by resolving indirect jump targets, which are a known challenge for static-only approaches. Also, efficiently recording control flow data can facilitate decoupled taint analysis [26, 63] to improve the overall taint analysis performance. However, the significantly high runtime overhead is the most obvious barrier to the complete collection of dynamic control flow, which further slows down the related security analysis tasks. On the other hand, the topic of efficient trace profiling has been well studied in the code generation and optimization area [95–97]. Unfortunately, these previous efforts cannot be directly applied to the obfuscated binary code. They either require an accurate control flow graph for path profile encoding or cannot work when continuous control flow information is required. Therefore, an efficient and complete binary code control flow collection tool is necessary to reverse engineering.

One of the byproduct of StraightTaint and TaintPipe projects is that we design a novel binary code control flow profiling tool, *BinCFP*, which allows efficient and complete binary code control flow profiling on ubiquitous multi-core platforms. Based on the logged data, we can construct a straight-line code trace for the further offline analysis. BinCFP is built on dynamic binary instrumentation so that it works with unmodified binary code. The core of BinCFP is a multi-threaded fast buffering scheme, which allows the application to continue executing while trace buffers are processed asynchronously. We also adopt an enhanced control flow profile structure to produce compact profile size. In addition, we apply a set of instrumentation optimizations to achieve better performance gains. Furthermore, BinCFP supports multi-threaded applications as well. We have developed BinCFP on top of Pin [17], a dynamic binary instrumentation platform. BinCFP relies on very lightweight logging of the execution control flow information to reconstruct a straight-line code later. We have evaluated BinCFP on a

number of applications such as SPEC2006 and obfuscated common utility programs. The performance experiments show that BinCFP imposes a small impact on application runtime performance and a considerable speedup over the previous approaches. Besides, BinCFP's control flow profile sizes are only about 49.2% that of conventional profiles. With optimization and compression, the profile size can be further lowered. Such experimental evidence shows that BinCFP can be applied for various software security applications. The details of BinCFP have been presented in Section 4.2.

To evaluate the application performance slowdown imposed exclusively by BinCFP, we have developed a simple tool to measure Pin's environment runtime overhead, which runs a program under Pin without any form of analysis ("nullpin" bar). Besides, we also compare BinCFP with other three profile formats. "BB_pc" bar refers the conventional 4-byte basic block profile, which records the entry address of each basic block. "CF_bit" indicate using a single bit to record the binary decision of a conditional jump. "DEP" refers the control flow profile scheme proposed by Zhao et al. [82].

Figure 4.11 shows the normalized execution time of running SPEC CINT2006 with *reference* workload. We expect these results can estimate the worst case. On average, BinCFP imposes a 2.97X slowdown to native execution, which is the lowest among the four profile formats. Due to the large profile size produced, BB_pc's application is often blocked for available free buffers. BinCFP outperforms DEP greatly in the test cases that use REP-prefixed instructions intensively, such as `h264ref` and `gcc`. CF_bit introduces as much as the 4.0X slowdown on average. The reason is CF_bit's instrumentation contains a number of conditional branches, which are hard to be inlined.

Then we evaluate BinCFP on four common Linux utilities, which represent three kinds of workload. `tar` is I/O bounded, whereas `bzip2` and `gzip` are CPU intensive programs. Between these two cases is *scp*. We use `tar` to archive and extract GNU

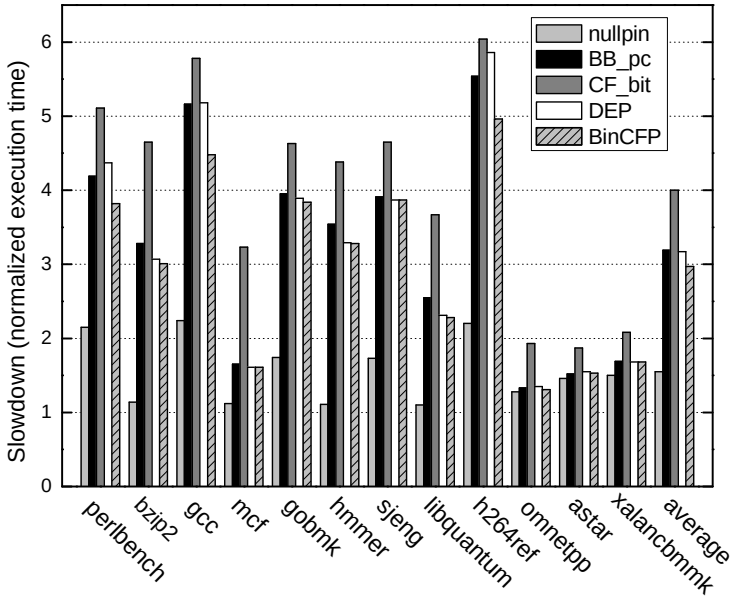

Figure 6.1. BinCFP slowdown on SPEC CPU2006.

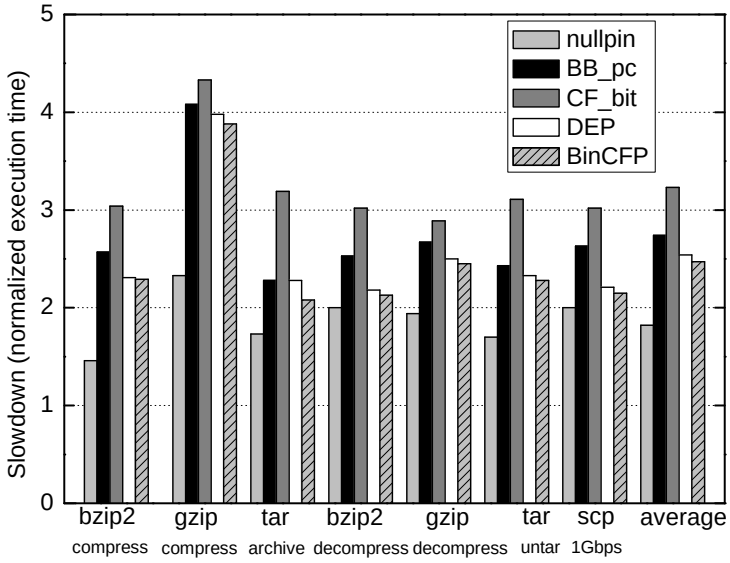

Figure 6.2. BinCFP slowdown on obfuscated common utilities.

Table 6.1. Different optimization or obfuscation options.

Obfuscator	Options
Loco	-freorder-blocks (reorder basic blocks) -funroll-loops (unroll loops) -finline-small-functions (inline functions)
Obf-LLVM	-mllvm -sub (instructions substitution) -mllvm -bcf (opaque predicate) -mllvm -fla (control flow flattening)

Core utilities 8.13 package (∼50MB). And then we apply `bzip2` and `gzip` to compress and decompress the archive file of Core utilities. For `scp`, we copy the archive file of Core utilities over 1Gbps link. Also, we obfuscate the tested programs by applying two obfuscators: Loco [98] and Obfuscator-LLVM [99]. Loco [98] is an obfuscation tool based on Diablo [100], a link-time optimizer. Diablo rewrites the binaries during link-time. Loco can obfuscate binaries by performing control flow flattening [101] and inserting opaque predicates [102]. Obfuscator-LLVM is a fork of the LLVM compilation suite [103] to provide code obfuscation options, and each option is implemented as an LLVM pass. The detailed obfuscation options are shown in Table 6.1. Note that several options can greatly obfuscate control flow graph, e.g., "-funroll-loops" unrolls loops, "-mllvm -bcf" inserts opaque predicates, and "-mllvm -fla" does control flow flattening. Therefore, the previous efforts on efficient path profiling [95–97] cannot be directly applied to these obfuscated binary code.

As shown in Figure 4.12, the experimental results are very similar to SPEC CPU2006. BinCFP imposes an average 2.47X slowdown to native execution, with a 1.30X speed up to CF_bit. Compared to nullpin, BinCFP only incurs a 1.36X slowdown. This evaluation shows that BinCFP supports the complete and efficient dynamic control flow collection of obfuscated binary code.

We also evaluate the profile size (uncompressed) introduced by the four different control flow profile formats: BB_pc, CF_bit, DEP, and BinCFP. We take the naive 4-byte

Table 6.2. Uncompressed trace profile size percentage (%).

Application	BB_pc	CF_bit	DEP	BinCFP
Utilities	100.0	21.9	56.6	51.6
SPEC CPU2006	100.0	22.6	53.0	47.8
Average	100.0	22.3	54.3	49.2

Table 6.3. Cryptography function detection time.

Algorithm	TaintPipe (s)	Temu (s)
TEA	3.8 (<1.1X)	15.2 (2.2X)
AES-CBC	12.3 (1.2X)	125.6 (3.8X)
Blowfish	4.5 (<1.1X)	21.4 (2.5X)
MD5	7.4 (<1.1X)	35.1 (2.6X)
SHA-1	8.8 (1.1X)	40.2 (3.3X)

profile (BB_pc) size as the baseline. Table 6.2 shows the average proportion related to BB_pc's profile size. The lower value in Table 6.2 means the size is more compact. On average, the relative size of BinCFP is only 49.2%, less than DEP's size by 5 percentages. It is worth pointing out that we see a significant profile size reduction for the *h264ref* benchmark, from 54.9% (DEP) to 24.6% (BinCFP). That is due to the intensive usage of `REP`-prefixed instructions. BinCFP optimizes this case (discussed in Section 4.2.2) and produces more compact profile size. CF_bit results in a much denser profile size with the 22.3% percentage. However, CF_bit instrumentation is hard to be optimized, which leads to the highest overhead in our evaluation. In summary, BinCFP represents a practical solution that balances the runtime performance and the profile size. Note that trace compression methods (e.g., VPC3 [104], Sequitur [105]) are also be applied on BinCFP. We can benefit from these new trace compression algorithms to further reduce the profile size.

6.2 Encoding Function Detection

Malware authors often use cryptography algorithms to encrypt malicious code, sensitive data, and communication. Detecting cryptography functions in the binary code facilitates malware analysis and forensics. Recent work leverages so called *avalanche effect*: each byte in the encrypted message is dependent on almost all bytes of input data or key [9, 106, 107]. One bit change in the input may cause significant change in the output. Therefore, the previous approaches quickly identify possible cryptography functions by observing the input-output dependency with multi-tag taint analysis. However, multi-tag dynamic taint analysis normally has to sacrifice more shadow memory and imposes much higher runtime overhead than single-tag dynamic taint analysis. Recall that our symbolic taint analysis supports multi-tag taint propagation transparently. We have carried out a preliminary experiment to detect such avalanche effects in binary code. We utilize the test case suite of Crypto++ library[1] and test 5 encryption algorithms. Each byte of the plain messages is labeled as a different taint tag. To this end, we taint each byte in the buffer where the data is stored. Our symbolic taint analysis tracks how the tainted inputs are propagated along the encryption algorithm execution. After that, we check each byte in the buffer of the `write()` system call to identify the high taint degree [9].

We compare TaintPipe with Temu [64], which supports multiple byte-to-byte taint propagation as well. The detection time is shown in Table 6.3. We also report the ratio of multi-tag's running time to single-tag's. The results show that TaintPipe is able to detect cryptographic functions with little additional overhead (less than 1.1X on average), while Temu's multi-tag propagation imposes a significant slowdown (2.9X to single-tag propagation on average). Table 6.3 shows the preliminary experiment result is

[1] http://www.cryptopp.com/

encouraging. We plan to apply our decoupled symbolic taint analysis to cryptography or encoding function detection in obfuscated binary code and malware.

6.3 Speeding Up Semantics-based Binary Diffing

Since most malware spread in binary form, the techniques to detect the difference between two binaries (*binary diffing*) have been widely applied to malware reverse engineering tasks. Conventional binary diffing tools identify syntactical differences such as instruction sequences, byte n-grams, and basic block hashing [108]. However, they can be easily evaded by various obfuscation methods [109–112]. The core method of the advanced semantics-based binary diffing [37,52,113,114] is to first identify semantically equivalent basic block pairs. It represent the inputs to the basic block as symbols and then simulates the effect of each instruction by updating the corresponding symbolic formula. The output of symbolic execution is a set of formulas that represent input-output relations of the basic block. After that, we try to find whether there is an equivalent mapping between two basic block output formulas. If yes, we conclude that two basic blocks are equivalent in semantics. Figure 6.3 presents a motivating example to illustrate how the semantics of a basic block is simulated by symbolic execution. The two basic blocks in Figure 6.3 are semantically equivalent, even though they have different x86 instructions (labeled as bold).

We take the inputs to the basic block as symbols and simulate the effect of each instruction by updated the corresponding symbolic formula. The output of symbolic execution is a set of formulas that represent input-output relations of the basic block. Now determining whether two basic blocks are equivalent in semantics boils down to find an equivalent mapping between output formulas. Note that due to obfuscation such as register renaming, basic blocks could use different registers or variables to provide the

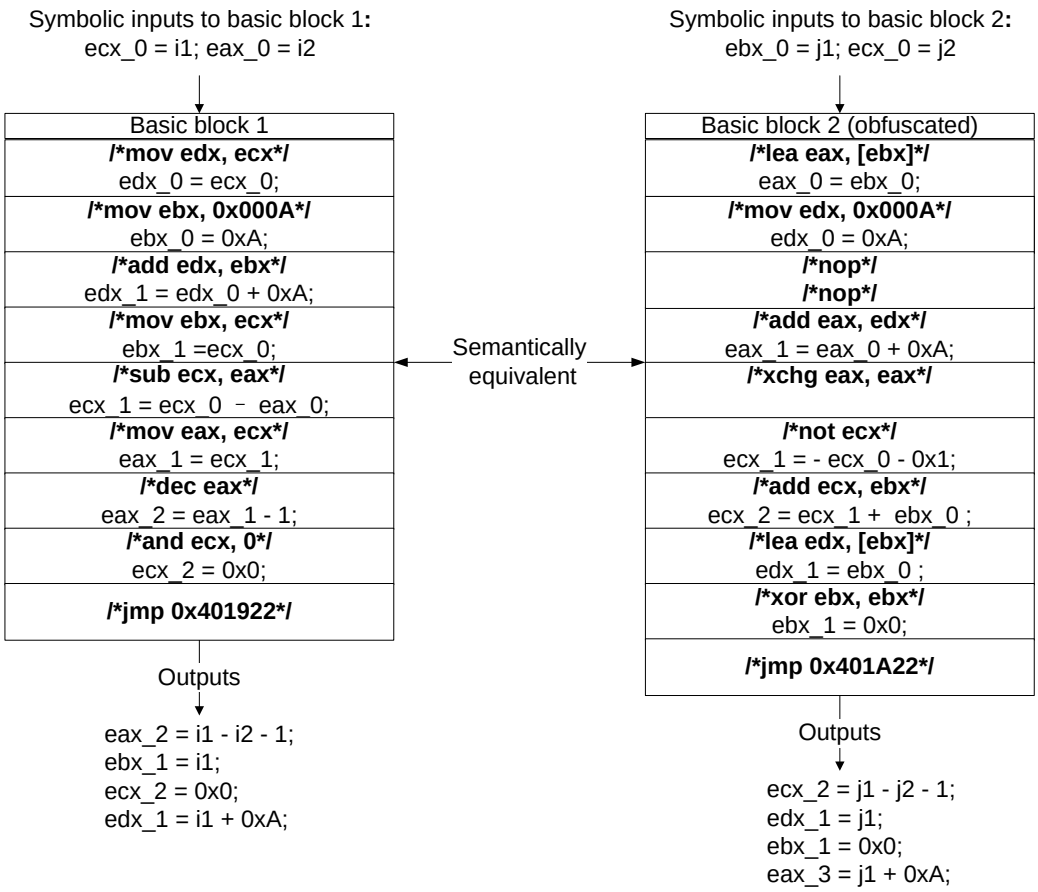

Figure 6.3. Example: basic block symbolic execution. The symbolic execution is performed based on IL (for brevity, we do not show the modification to the EFLAGS bits).

Query result	eax_3	ebx_1	ecx_2	edx_1
eax_2	false		**true**	false
ebx_1	false		false	**true**
ecx_2		**constant (0)**		
edx_1	**true**		false	false

Figure 6.4. Output formulas equivalence query results.

same functionality. As a result, current approaches exhaustively try all possible pairs to find if there exists a bijective mapping between output formulas. Figure 6.4 shows such formulas mapping attempt for the output formulas shown in Figure 6.3. The variables shown in the leftmost column are from basic block 1 in Figure 6.3; while the variables in the upmost row are from basic block 2. The “true” or “false” indicates the result of

equivalence checking, such as whether $edx\_1 = eax\_3$. After 10 times comparisons, we identify a perfect matched permutation and therefore conclude that these two basic blocks are truly equivalent.

However, semantics-based binary diffing suffers from significant performance slowdown, hindering it from analyzing large numbers of malware samples [115, 116]. We look into the overhead imposed by semantics-based binary diffing and find that one major factor dominates the cost; that is, the high number of invocations of constraint solver. Recall that current approaches check all possible permutations of output formulas mapping. The constraint solver will be invoked every time when verifying the equivalence of formulas. For example, two basic blocks in Figure 6.3 have three symbolic formulas and one constant value, respectively. As shown in Figure 6.4, We have to employ constraint solver at most nine times to find an equivalent mapping between the three output formulas. Too frequently calling constraint solver incurs a significant performance penalty.

To reduce the number of basic blocks to be compared, we utilize multi-tag taint analysis to reduce the number of possible matches. For example, we can assign different taint tags to various fields of the input or system call output values. Basic block pairs that share the same taint tags have the high priority to match. In this way, the set of matching candidates of each basic block drops from the set of all basic blocks in a program (in terms of thousands or tens of thousands) to just a few basic blocks on a particular execution trace with the same taint tags. We compare `thttpd` and `gzip` with their different versions. We first assigns different taint tags to various parts or fields of the input and then compare execution traces to find semantically equivalent basic blocks. Out of the basic blocks that are successfully matched, we count the number of them that actually contain the same taint tags. Results (see Table 6.4 and Table 6.5) show that

more than 34% and 67% of the matched basic blocks in `thttpd` and `gzip`, respectively, contain the same taint tags. This shows that 1) multi-tag taint analysis is effective in helping to identify basic block matchings, as a large number of these matchings do contain the same taint representation; 2) even though many basic blocks are not tainted by our limited number of program inputs, their neighbors are tainted in most cases and the tainted neighbors help matchings to be identified.

thttpd-	2.20	2.20c	2.21	2.25
2.19	34.8%	38.2%	39.9%	37.4%

Table 6.4. Percentage of matched basic blocks with the same taint tags (thttpd).

gzip-	1.3.12	1.3.13	1.40
1.2.4	67.9%	72.2%	72.6%

Table 6.5. Percentage of matched basic blocks with the same taint tags (gzip).

When comparing two basic blocks, current approaches exhaustively try all possible pairs to find if there exists a bijective mapping between output formulas. This permutation matching leads to a performance bottleneck. Multi-tag taint propagation can improve binary diffing in two ways: 1). We can first compare output variables with the same taint tags. 2). We can compare instruction slice that only is tainted by the same taint color. As we have shown in Section 4.5.4, traditional multi-tag taint analysis is expensive. By contrast, our decoupled symbolic multi-tag taint analysis introduces small additional overhead. Semantics-based binary diffing and decoupled symbolic multi-tag taint analysis can seamlessly weave together and amplify each other's benefit. By applying decoupled symbolic multi-tag taint analysis, our approach significantly increases the effectiveness and efficiency of semantics-based binary diffing.

Chapter 7
Discussion and Future Work

In this chapter, we discuss some additional problems of our proposed techniques and possible future work.

7.1 Discussion

StraightTaint is a prototype to demonstrate that completely decoupling dynamic taint analysis is feasible. The performance of online logging and offline taint analysis can be further improved. StraightTaint's offline taint analysis is as fast as, but not faster than, DTA on average, but not faster, since in StraightTaint the semantics of taint operations are simulated. On the other hand, StraightTaint offline analysis is full-featured and performing multi-tag and bit-level taint analysis is natural, with very low extra overhead, while DTA typically incurs much higher extra overhead for multi-tag or bit-level taint analysis.

Since TaintPipe's pipelining design leads to an asynchronous taint check, TaintPipe may detect a violation of taint policy after the real attack happens. One possible solution is to provide *synchronous* policy enforcement at critical points (e.g., indirect jump and system call sites). In that case, we can explicitly suspend the application thread, and

wait for the worker threads to complete. As TaintPipe may perform symbolic taint analysis when explicit taint states are not available, TaintPipe exhibits similar limitations as symbolic execution of binaries. TaintPipe recovers the straight-line code by logging basic block entry address. However, with malicious self-modifying code, the entry address may not uniquely identify a code block. To address this issue, we can augment TaintPipe by logging the real executed instructions at the expense of runtime performance overhead. Our focus is to demonstrate the feasibility of pipelined symbolic taint analysis. We have not fully optimized the symbolic taint analysis part which we believe can be greatly improved in terms of performance based on our current prototype. Currently TaintPipe requires large share memory to reduce communication overhead between different pipeline stages. Therefore, our approach is more suitable for large servers with sufficient memory.

Currently, BinCFP's control flow profile encodes basic block entry addresses. However, the entry address may not uniquely identify self-modifying code blocks. To address this issue, we can configure BinCFP to record the real executed instructions. BinCFP spawns worker threads in the same process running both Pin and the application. To increase isolation, we plan to switch the worker threads to different processes.

It is important to discuss how precise our symbolic taint analysis can achieve. As we have demonstrated, StraightTaint supports full-featured taint propagation strategies, which are quite robust to trace explicit information flow in the context of obfuscated binary analysis. Our symbolic taint analysis leads to much more accurate taint propagation results than conventional dynamic taint analysis tools. However, we have identified two significant challenges that can prevent our approach from further improving the accuracy. First, StraightTaint's conditional tainting represents the causal relationships between sources and sinks with taint logic constraints, which will be solved later by a constraint

solver to map new inputs. One challenge here is a taint logic constraint could become too complicated. Subject to computing resources (e.g., memory and CPU) and the inherent undecidability of the problem, a complicated formula could be hard or infeasible to solve. Second, malware writers could evade dynamic taint analysis through control dependence and implicit flows, such as pointer tainting [117]. Cavallaro et al. [118] have demonstrated that mitigating such evasion attacks will lead to an increase in false positives as well. We will continue to address these challenges and improve our symbolic taint analysis accuracy.

7.2 Future Work

Currently, the upper bound of online logging performance that we can achieve is restricted by Pin's environment runtime overhead. One of our future work is to leverage the advanced development toolkits such as Uroboros [67, 119] so that we can insert the taint tracking code directly at the disassemble code and then compile it to the binary code again. In this way, we can remove DBI's environment overhead. One future work to speed up offline taint analysis is to construct a recompilable straight-line program from execution trace. As a result, we can apply another round of DTA directly on the straight-line program. Currently, StraightTaint works on sequential programs. To support taint analysis for multi-threaded programs, we have to carefully handle the complicated inter-thread taint propagation, such as concurrent accesses to shared locations and corresponding taint tag updates. We plan to explore these directions in future. The collected data during online logging for some long running program are still too large to be saved up for offline analysis. Another direction we plan to explore is decoupling taint analysis via parallelization, which performs taint analysis on-the-fly without the log storage.

TaintPipe spawns worker threads in the same process of running both Pin and the

application. In the future, we plan to replace the worker threads with different processes to increase isolation. Recent work MAYHEM [89] proposes an advanced index-based memory model to deal with symbolic memory index. We plan to extend our symbolic memory index handling in the future. As our taint analysis engine simulates the semantics of taint operations, the speed of taint analysis is slow. One future direction is to execute concrete taint analysis natively like micro execution [85] and switch to the interpretation-style when performing symbolic taint analysis.

An interesting future work for BinCFP is to study efficient memory references profiling for the obfuscated binary code. Since memory references profiling size is much larger than control flow profiling size, we have to carefully design trace profile structure. Because emerging platforms (e.g., mobile phones and embedded systems) have been widely employed in our daily lives to serve critical functions, , the security of emerging platform becomes extremely important. Another direction I would like to explore is to develop decoupled taint analysis to reason security and safety properties for emerging platforms.

Chapter 8
Conclusion

Taint analysis has a wide variety of compelling applications in security tasks, from software attack detection to data lifetime analysis. Static taint analysis propagates taint values following all possible paths with no need for concrete execution, but is generally less accurate than dynamic analysis. Unfortunately, the high performance penalty incurred by dynamic taint analyses makes its deployment impractical in production systems. To ameliorate this performance bottleneck, recent research efforts aim to decouple data flow tracking logic from program execution. We continue this line of research in this thesis and propose *pipelined symbolic taint analysis*, a novel technique for decoupling and parallelizing taint analysis to take advantage of ubiquitous multi-core platforms.

In this thesis, we first presented StraightTaint, a technique for completely decoupling dynamic taint analysis for offline symbolic taint analysis. Unlike previous approaches, StraightTaint does not rely on complete runtime values or inputs, which enables very lightweight logging and much lower online execution slowdown. StraightTaint can also support full-featured, multi-tag, and bit-level taint analysis with low extra overhead. We have evaluated StraightTaint on a number of applications such as utility programs, SPEC2006, and real-life software vulnerabilities. The results show that StraightTaint can

rival dynamic taint analysis at a similar level of precision, but with a much lower online execution slowdown and more flexible functionalities. The performance experiments demonstrate that StraightTaint can speed up application runtime performance 3.32 times on a set of utility programs, and 3.25 times on SPEC2006, respectively. Such experimental evidence indicates that StraightTaint can be applied to speed up various *ex post facto* security applications with full-featured offline taint analysis.

We then presented TaintPipe, a novel tool for pipelining dynamic taint analysis with segmented symbolic taint analysis. Different from previous parallelization work on taint analysis, TaintPipe uses a pipeline style that relies on straight-line code with very few runtime values, enabling lightweight online logging and much lower runtime overhead. Instead of waiting for the explicit taint state, each worker thread can start running symbolic taint analysis very early. In this way, TaintPipe has much smaller delay with the application execution. We have evaluated TaintPipe on a number of benign and malicious programs. The results show that TaintPipe rivals conventional inlined dynamic taint analysis in precision, but with a much lower online execution slowdown. The performance experiments indicate that TaintPipe can speed up dynamic taint analysis by 2.43 times on a set of common utilities and 2.38 times on SPEC2006, respectively. Such experimental evidence demonstrates that TaintPipe is both efficient and effective to be applied in real production environments.

The multi-threaded fast buffering scheme we designed for the StraightTaint and TaintPipe projects has been applied to binary code control flow profiling. We have developed *BinCFP*, an efficient multi-threaded binary code control flow profiling tool by taking advantage of ubiquitous multi-core platforms. Unlike previous approaches, BinCFP does not rely on fine-grained static analysis, which enables broad applications in speeding up binary code analysis such as hybrid disassembly and decoupled taint analysis.

The experimental results on SPEC2006 and obfuscated common utility programs are encouraging, which shows a considerable speedup over the previous work. Because of the dynamic taint analysis decoupling style, both StraightTaint and TaintPipe are particularly well suited for *ex post facto* security analysis tasks, such as computer forensic analysis and reverse engineering. To make our techniques applicable to a broader range of security applications, we have integrated the decoupled symbolic multi-tag taint analysis into the tasks of encoding function detection and speeding up semantics-based binary diffing. With our techniques, we are able to detect cryptography functions in binary code with little additional overhead and significantly reduce the number of possible basic block to be compared.

In summary, our research work demonstrates that pipelined symbolic taint analysis is a compelling method to complement existing expensive runtime security analysis. In future, we plan to further reduce offline taint analysis overhead, support taint analysis for multi-threaded programs, and develop decoupled taint analysis for mobile platforms.

Bibliography

[1] DAVID BRUMLEY, "Exploits: Buffer Overflows and Format String Attacks," 2013 UCSD DARPA Presentation.

[2] MCAFEE LABS , "2016 Threats Predictions," http://www.mcafee.com/us/resources/reports/rp-threats-predictions-2016.pdf.

[3] EBIOSCIENCE, "Cell Biology Dyes and Staining Reagents," https://www.ebioscience.com/media/pdf/literature/cell-biology-dyes-reagents-brochure.pdf.

[4] NEWSOME, J. and D. SONG (2005) "Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software," in *Proceedings of the Network and Distributed System Security Symposium (NDSS'05)*.

[5] XU, W., S. BHATKAR, and R. SEKAR (2006) "Taint-enhanced Policy Enforcement: A Practical Approach to Defeat a Wide Range of Attacks," in *Proceedings of the 15th Conference on USENIX Security Symposium*.

[6] YIP, A., X. WANG, N. ZELDOVICH, and M. F. KAASHOEK (2009) "Improving application security with data flow assertions," in *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles (SOSP'09)*.

[7] VACHHARAJANI, N., M. J. BRIDGES, J. CHANG, R. RANGAN, G. OTTONI, J. A. BLOME, G. A. REIS, M. VACHHARAJANI, and D. I. AUGUST (2004) "RIFLE: An Architectural Framework for User-Centric Information-Flow Security," in *Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'37)*.

[8] ZHU, D. Y., J. JUNG, D. SONG, T. KOHNO, and D. WETHERALL (2011) "TaintEraser: protecting sensitive data leaks using application-level taint tracking," *ACM SIGOPS Operating Systems Review*, **45**, pp. 142–154.

[9] CABALLERO, J., P. POOSANKAM, S. MCCAMANT, D. BABI Ć, and D. SONG (2010) "Input Generation via Decomposition and Re-stitching: Finding Bugs

in Malware," in *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS'10)*.

[10] YIN, H., D. S. AMD M. EGELE, C. KRUEGEL, and E. KIRDA (2007) "Panorama: Capturing system-wide information flow for malware detection and analysis," in *ACM Conference on Computer and Communications Security (CCS'07)*.

[11] ARZT, S., S. RASTHOFER, C. FRITZ, E. BODDEN, A. BARTEL, J. KLEIN, Y. LE TRAON, D. OCTEAU, and P. MCDANIEL (2014) "FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'14)*.

[12] WANG, X., Y.-C. JHI, S. ZHU, and P. LIU (2008) "STILL: Exploit Code Detection via Static Taint and Initialization Analyses," in *Proceedings of the 24th Annual Computer Security Applications Conference (ACSAC'08)*.

[13] RAWAT, S., L. MOUNIER, and M.-L. POTET (2011), "Static Taint-Analysis on Binary Executables," `http://stator.imag.fr/w/images/2/21/Laurent_Mounier_2013-01-28.pdf`.

[14] SCHWARTZ, E. J., T. AVGERINOS, and D. BRUMLEY (2010) "All You Ever Wanted to Know About Dynamic Taint Analysis and Forward Symbolic Execution," in *Proceedings of the IEEE Symposium on Security and Privacy (SP'10)*.

[15] CLAUSE, J., W. P. LI, and A. ORSO (2007) "Dytan: A Generic Dynamic Taint Analysis Framework," in *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'07)*.

[16] QIN, F., C. WANG, Z. LI, H. SEOP KIM, Y. ZHOU, and Y. WU (2006) "LIFT: A Low-Overhead Practical Information Flow Tracking System for Detecting Security Attacks," in *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*.

[17] LUK, C.-K., R. COHN, R. MUTH, H. PATIL, A. KLAUSER, G. LOWNEY, S. WALLACE, V. J. REDDI, and K. HAZELWOOD (2005) "Pin: building customized program analysis tools with dynamic instrumentation," in *Proceedings of the ACM SIGPLAN conference on Programming language design and implementation (PLDI'05)*.

[18] BELLARD, F. (2005) "QEMU, a Fast and Portable Dynamic Translator," in *Proceedings of the 2005 USENIX Annual Technical Conference (ATC'05)*.

[19] KEMERLIS, V. P., G. PORTOKALIDIS, K. JEE, and A. D. KEROMYTIS (2012) "libdft: Practical Dynamic Data Flow Tracking for Commodity Systems," in

Proceedings of the 8th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE'12).

[20] RUWASE, O., S. CHEN, P. B. GIBBONS, and T. C. MOWRY (2010) "Decoupled Lifeguards: Enabling Path Optimizations for Dynamic Correctness Checking Tools," in *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'10)*.

[21] CHENG, W., Q. ZHAO, B. YU, and S. HIROSHIGE (2006) "TaintTrace: Efficient Flow Tracing with Dynamic Binary Rewriting," in *Proceedings of the 11th IEEE Symposium on Computers and Communications (ISCC'06)*.

[22] NIGHTINGALE, E. B., D. PEEK, P. M. CHEN, and J. FLINN (2008) "Parallelizing Security Checks on Commodity Hardware," in *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'08)*.

[23] RUWASE, O., P. B. GIBBONS, T. C. MOWRY, V. RAMACHANDRAN, S. CHEN, M. KOZUCH, and M. RYAN (2008) "Parallelizing Dynamic Information Flow Tracking Lifeguards," in *Proceedings of the 20th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA'08)*.

[24] NAGARAJAN, V., H.-S. KIM, Y. WU, and R. GUPTA (2008) "Dynamic information flow tracking on multicores," in *Proceedings of the Workshop on Interaction between Compilers and Computer Architectures*.

[25] ERMOLINSKIY, A., S. KATTI, S. SHENKER, L. L. FOWLER, and M. MCCAULEY (2010) *Towards Practical Taint Tracking*, *Tech. rep.*, EECS Department, University of California, Berkeley.

[26] JEE, K., V. P. KEMERLIS, A. D. KEROMYTIS, and G. PORTOKALIDIS (2013) "ShadowReplica: Efficient Parallelization of Dynamic Data Flow Tracking," in *Proceedings of the ACM SIGSAC conference on Computer & communications security (CCS'13)*.

[27] CUI, B., F. WANG, T. GUO, and G. DONG (2015) "A practical off-line taint analysis framework and its application in reverse engineering of file format," *Computers & Security*, **51**(C).

[28] WHELAN, R., T. LEEK, and D. KAELI (2013) "Architecture-Independent dynamic information flow tracking," in *Proceedings of the 22nd international conference on Compiler Construction (CC'13)*.

[29] WAND, C.-W. and S. W. SHIEH (2015) "SWIFT: Decoupled System-Wide Information Flow Tracking and its Optimizations," *Journal Of Information Science and Engineering*, **31**(4).

[30] HENDERSON, A., A. PRAKASH, L. K. YAN, X. HU, X. WANG, R. ZHOU, and H. YIN (2014) "Make It Work, Make It Right, Make It Fast: Building a Platform-neutral Whole-system Dynamic Binary Analysis Platform," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis (ISSTA'14)*.

[31] YADEGARI, B. and S. DEBRAY (2014) "Bit-Level Taint Analysis," in *Proceedings of the 14th IEEE International Working Conference on Source Code Analysis and Manipulation*.

[32] CABALLERO, J., H. YIN, Z. LIANG, and D. SONG (2007) "Polyglot: Automatic Extraction of Protocol Message Format Using Dynamic Binary Analysis," in *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS'07)*.

[33] BRUMLEY, D., I. JAGER, T. AVGERINOS, and E. J. SCHWARTZ (2011) "BAP: A Binary Analysis Platform," in *Proceedings of the 23rd international conference on computer aided verification (CAV'11)*.

[34] LEE, K. H., X. ZHANG, and D. XU (2013) "High Accuracy Attack Provenance via Binary-based Execution Partition," in *Proceedings of the 20th Network and Distributed System Security Symposium (NDSS'13)*.

[35] XIAO, G., J. WANG, P. LIU, J. MING, and D. WU (2016) "Program-object Level Data Flow Analysis with Applications to Data Leakage and Contamination Forensics," in *Proceedings of the 6th ACM Conference on Data and Application Security and Privacy (CODASPY'16)*.

[36] KRISHNAN, S., K. Z. SNOW, and F. MONROSE (2010) "Trail of Bytes: Efficient Support for Forensic Analysis," in *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS'10)*.

[37] MING, J., M. PAN, and D. GAO (2012) "iBinHunt: Binary Hunting with Inter-Procedural Control Flow," in *Proceedings of the 15th Annual International Conference on Information Security and Cryptology (ICISC'12)*.

[38] YADEGARI, B., B. JOHANNESMEYER, B. WHITELY, and S. DEBRAY (2015) "A Generic Approach to Automatic Deobfuscation of Executable Code," in *Proceedings of the 36th IEEE Symposium on Security and Privacy*.

[39] PAPPAS, V., V. P. KEMERLIS, A. ZAVOU, M. POLYCHRONAKIS, and A. D. KEROMYTIS (2013) "CloudFence: Data Flow Tracking as a Cloud Service," in *Proceedings of the 16th International Symposium on Research in Attacks, Intrusions and Defenses (RAID'13)*.

[40] RAWAT, S., L. MOUNIER, and M.-L. POTET (2011), "Static Taint-Analysis on Binary Executables," http://stator.imag.fr/w/images/2/21/Laurent_Mounier_2013-01-28.pdf.

[41] BODDEN, E. (2012) "Inter-procedural data-flow analysis with ifds/ide and soot," in *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program analysis*, ACM, pp. 3–8.

[42] KLIEBER, W., L. FLYNN, A. BHOSALE, L. JIA, and L. BAUER (2014) "Android taint flow analysis for app sets," in *Proceedings of the 3rd ACM SIGPLAN International Workshop on the State of the Art in Java Program Analysis*, ACM, pp. 1–6.

[43] LINN, C. and S. DEBRAY (2003) "Obfuscation of Executable Code to Improve Resistance to Static Disassembly," in *Proceedings of the 10th ACM Conference on Computer and Communications Security (CCS'03)*.

[44] SLOWINSKA, A., T. STANCESCU, and H. BOS (2011) "Howard: A Dynamic Excavator for Reverse Engineering Data Structures," in *Proceedings of the 2011 Network and Distributed System Security Symposium (NDSS'11)*.

[45] WONDRACEK, G., P. M. COMPARETTI, C. KRUEGEL, and E. KIRDA (2008) "Automatic Network Protocol Analysis," in *Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS'08)*.

[46] CABALLERO, J., P. POOSANKAM, C. KREIBICH, and D. SONG (2009) "Dispatcher: Enabling Active Botnet Infiltration using Automatic Protocol Reverse-Engineering," in *Proceedings of the 16th ACM Conference on Computer and Communication Security (CCS'09)*.

[47] XU, M., V. MALYUGIN, J. SHELDON, G. VENKITACHALAM, and B. WEISSMAN (2007) "ReTrace: Collecting Execution Trace with Virtual Machine Deterministic Replay," in *Proceedings of the Workshop on Modeling, Benchmarking and Simulation*.

[48] ENCK, W., P. GILBERT, B.-G. CHUN, L. P. COX, J. JUNG, P. MCDANIEL, and A. N. SHETH (2010) "TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones," in *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI'10)*.

[49] JHI, Y.-C., X. WANG, X. JIA, S. ZHU, P. LIU, and D. WU (2011) "Value-Based Program Characterization and Its Application to Software Plagiarism Detection," in *Proceedings of the ACM/IEEE 33rd International Conference on Software Engineering (ICSE'11), Software Engineering in Practice Track*.

[50] ZHANG, F., Y.-C. JHI, D. WU, P. LIU, and S. ZHU (2012) "A First Step Towards Algorithm Plagiarism Detection," in *Proceedings of the 2012 International Symposium on Software Testing and Analysis (ISSTA'12)*.

[51] ZHANG, F., D. WU, P. LIU, and S. ZHU (2014) "Program Logic Based Software Plagiarism Detection," in *Proceedings of the 25th annual International Symposium on Software Reliability Engineering (ISSRE'14)*.

[52] LUO, L., J. MING, D. WU, P. LIU, and S. ZHU (2014) "Semantics-based Obfuscation-resilient Binary Code Similarity Comparison with Applications to Software Plagiarism Detection," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'14)*.

[53] JHI, Y.-C., X. JIA, X. WANG, S. ZHU, P. LIU, and D. WU (2015) "Program Characterization Using Runtime Values and Its Application to Software Plagiarism Detection," *IEEE Transactions on Software Engineering*, **41**(9), pp. 925–943.

[54] MING, J., F. ZHANG, D. WU, P. LIU, and S. ZHU (2016) "Deviation-Based Obfuscation-Resilient Program Equivalence Checking with Application to Software Plagiarism Detection," *IEEE Transactions on Reliability*, **65**(4).

[55] LAM, L. C. and T.-C. CHIUEH (2006) "A general dynamic information flow tracking framework for security applications," in *Proceedings of the 22nd Annual Computer Security Applications Conference (ACSAC'06)*, IEEE, pp. 463–472.

[56] CHANG, W., B. STREIFF, and C. LIN (2008) "Efficient and extensible security enforcement using dynamic data flow analysis," in *Proceedings of the 15th ACM conference on Computer and communications security*, ACM, pp. 39–50.

[57] SUH, G. E., J. W. LEE, D. ZHANG, and S. DEVADAS (2004) "Secure Program Execution via Dynamic Information Flow Tracking," in *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'04)*.

[58] CRANDALL, J. R. and F. T. CHONG (2004) "Minos: Control data attack prevention orthogonal to memory model," in *Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture*.

[59] BOSMAN, E., A. SLOWINSKA, and H. BOS (2011) "Minemu: The World's Fastest Taint Tracker," in *Proceedings of the 14th International Symposium on Recent Advances in Intrusion Detection (RAID'11)*.

[60] JEE, K., G. PORTOKALIDIS, V. P. KEMERLIS, S. GHOSH, D. I. AUGUST, and A. D. KEROMYTIS (2012) "A General Approach for Efficiently Accelerating Software-based Dynamic Data Flow Tracking on Commodity Hardware." in

Proceedings of the Network and Distributed System Security Symposium (NDSS'12).

[61] CHABBI, M., S. PERIYANAYAGAM, G. ANDREWS, and S. DEBRAY (2007) *Efficient Dynamic Taint Analysis Using Multicore Machines*, *Tech. rep.*, The University of Arizona.

[62] CHEN, S., P. B. GIBBONS, M. KOZUCH, and T. C. MOWRY (2011) "Log-based Architectures: Using Multicore to Help Software Behave Correctly," *ACM SIGOPS Operating Systems Review*, **45**(1), pp. 84–91.

[63] MING, J., D. WU, G. XIAO, J. WANG, and P. LIU (2015) "TaintPipe: Pipelined Symbolic Taint Analysis," in *Proceedings of the 24th USENIX Security Symposium (USENIX Security'15)*.

[64] YIN, H. and D. SONG (2010) *TEMU: Binary Code Analysis via Whole-System Layered Annotative Execution*, *Tech. Rep. UCB/EECS-2010-3*, EECS Department, University of California, Berkeley.

[65] MING, J., D. WU, J. WANG, G. XIAO, and P. LIU (2016) "StraightTaint: Decoupled Offline Symbolic Taint Analysis," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE'16)*.

[66] LINN, C. and S. DEBRAY (2003) "Obfuscation of Executable Code to Improve Resistance to Static Disassembly," in *Proceedings of the 10th ACM Conference on Computer and Communications Security (CCS'03)*.

[67] WANG, S., P. WANG, and D. WU (2015) "Reassembleable Disassembling," in *Proceedings of the 24th USENIX Security Symposium (USENIX Security'15)*.

[68] CHOW, J., T. GARFINKEL, and P. M. CHEN (2008) "Decoupling dynamic program analysis from execution in virtual environments," in *Proceedings of the 2008 USENIX Annual Technical Conference (ATC'08)*.

[69] CHEN, Y. and H. CHEN (2013) "Scalable Deterministic Replay in a Parallel Full-system Emulator," in *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'13)*.

[70] PATIL, H., C. PEREIRA, M. STALLCUP, G. LUECK, and J. COWNIE (2010) "PinPlay: A Framework for Deterministic Replay and Reproducible Analysis of Parallel Programs," in *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO'10)*.

[71] LIVSHITS, B., A. V. NORI, S. K. RAJAMANI, and A. BANERJEE (2009) "Merlin: Specification Inference for Explicit Information Flow Problems," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'09)*.

[72] MYERS, A. C. (1999) "JFlow: Practical mostly-static information flow control," in *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, ACM, pp. 228–241.

[73] KING, D., T. JAEGER, S. JHA, and S. A. SESHIA (2008) "Effective blame for information-flow violations," in *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, ACM, pp. 250–260.

[74] BOUNIMOVA, E., P. GODEFROID, and D. MOLNAR (2013) "Billions and Billions of Constraints: Whitebox Fuzz Testing in Production," in *Proceedings of the International Conference on Software Engineering (ICSE'13)*.

[75] CADAR, C., V. GANESH, P. PAWLOWSKI, D. DILL, and D. ENGLER. (2006) "EXE:Automatically generating inputs of death," in *Proceedings of the ACM Conference on Computer and Communications Security (CCS'06)*.

[76] CADAR, C., D. DUNBAR, and D. ENGLER. (2008) "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI'08)*.

[77] GODEFROID, P., M. Y. LEVIN, and D. MOLNAR. (2008) "Automated Whitebox Fuzz Testing," in *Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS'08)*.

[78] YANG, G., C. S. PĂSĂREANU, and S. KHURSHID (2012) "Memoized Symbolic Execution," in *Proceedings of the 2012 International Symposium on Software Testing and Analysis (ISSTA'12)*.

[79] PHAM, V.-T., W. B. NG, K. RUBINOV, and A. ROYCHOUDHURY (2015) "Hercules: Reproducing Crashes in Real-world Application Binaries," in *Proceedings of the 37th International Conference on Software Engineering (ICSE'15)*.

[80] CHEN, S., M. KOZUCH, T. STRIGKOS, B. FALSAFI, P. B. GIBBONS, T. C. MOWRY, V. RAMACHANDRAN, O. RUWASE, M. RYAN, and E. VLACHOS (2008) "Flexible Hardware Acceleration for Instruction-Grain Program Monitoring," in *Proceedings of the 35th Annual International Symposium on Computer Architecture (ISCA'08)*.

[81] CHOW, J., T. GARFINKEL, and P. M. CHEN (2008) "Decoupling dynamic program analysis from execution in virtual environments," in *Proceedings of the USENIX Annual Technical Technical Conference (ATC'08)*.

[82] ZHAO, Q., J. E. SIM, L. RUDOLPH, and W.-F. WONG (2006) "DEP: Detailed Execution Profile," in *Proc. of the 15th International Conf. on Parallel Architectures and Compilation Techniques (PACT'06)*.

[83] BRUENING, D., T. GARNETT, and S. AMARASINGHE (2003) "An Infrastructure for Adaptive Dynamic Optimization," in *Proceedings of the international symposium on code generation and optimization (CGO'03)*.

[84] RENIERIS, M., S. RAMAPRASAD, and S. P. REISS (2005) "Arithmetic Program Paths," in *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE-13)*.

[85] GODEFROID, P. (2014) "Micro Execution," in *Proceedings of the 36th International Conference on Software Engineering (ICSE'14)*.

[86] MOURA, L. D. and N. BJØRNER (2008) "Z3: an efficient SMT solver," in *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*.

[87] KING, S. T. and P. M. CHEN (2003) "Backtracking intrusions," in *Proceedings of the 9th ACM symposium on Operating systems principles (SOSP'03)*.

[88] ZHAO, Q., I. CUTCUTACHE, and W.-F. WONG (2009) "PiPA: Pipelined Profiling and Analysis on Multi-core Systems," in *Proceedings of the International Symposium on Code Generation and Optimization (CGO'08)*.

[89] CHA, S. K., T. AVGERINOS, A. REBERT, and D. BRUMLEY (2012) "Unleashing Mayhem on Binary Code," in *Proceedings of the 2012 IEEE Symposium on Security and Privacy*.

[90] BALAKRISHNAN, G. and T. REPS (2010) "WYSINWYX: What You See Is Not What You eXecute," *ACM transactions on programming languages and systems*, **32**(6).

[91] CHRISTODORESCU, M., S. JHA, and C. KRUEGEL (2007) "Mining Specifications of Malicious Behavior," in *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering (ESEC-FSE'07)*.

[92] JEE, K., G. PORTOKALIDIS, V. P. KEMERLIS, S. GHOSH, D. I. AUGUST, and A. D. KEROMYTIS (2012) "A General Approach for Efficiently Accelerating Software-based Dynamic Data Flow Tracking on Commodity Hardware," in *Proceedings of the 19th Internet Society (ISOC) Symposium on Network and Distributed System Security (NDSS'12)*.

[93] CABALLERO, J., N. M. JOHNSON, S. MCCAMANT, and D. SONG (2010) "Binary Code Extraction and Interface Identification for Security Applications," in *Proceedings of 17th Annual Network and Distributed System Security Symposium (NDSS'10)*.

[94] UDUPA, S. K., S. K. DEBRAY, and M. MADOU (2005) "Deobfuscation: Reverse engineering obfuscated code," in *Proceedings of the 12th Working Conference on Reverse Engineering (WCRE'05)*.

[95] BALL, T. and J. R. LARUS (1996) "Efficient Path Profiling," in *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture (MICRO'29)*.

[96] BOND, M. D. and K. S. MCKINLEY (2005) "Practical Path Profiling for Dynamic Optimizers," in *Proceedings of the International Symposium on Code Generation and Optimization (CGO'05)*.

[97] JOSHI, R., M. D. BOND, and C. ZILLES (2004) "Targeted Path Profiling: Lower Overhead Path Profiling for Staged Dynamic Optimization Systems," in *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization (CGO'04)*.

[98] MADOU, M., L. VAN PUT, and K. DE BOSSCHERE (2006) "Loco: An Interactive Code (De)Obfuscation tool," in *Proceedings of ACM SIGPLAN 2006 Workshop on Partial Evaluation and Program Manipulation (PEPM'06)*.

[99] JUNOD, P., J. RINALDINI, J. WEHRLI, and J. MICHIELIN (2015) "Obfuscator-LLVM - Software Protection for the Masses," in *Proceedings of the 1st International Workshop on Software Protection (SPRO'15)*.

[100] "Diablo Is A Better Link-time Optimizer," [online]. Available: `http://diablo.elis.ugent.be/`.

[101] WANG, C., J. HILL, J. C. KNIGHT, and J. W. DAVIDSON (2001) "Protection of Software-Based Survivability Mechanisms," in *Proceedings of the 2001 International Conference on Dependable Systems and Networks (DSN'01)*.

[102] MING, J., D. XU, L. WANG, and D. WU (2015) "LOOP: Logic-Oriented Opaque Predicate Detection in Obfuscated Binary Code," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS'15)*.

[103] LATTNER, C. and V. ADVE (2004) "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *Proceedings of the International Symposium on Code Generation and Optimization (CGO'04)*.

[104] BURTSCHER, M. (2004) "VPC3: a fast and effective trace-compression algorithm," in *Proceedings of the joint International Conference on Measurement and Modeling of Computer Systems*.

[105] NEVILL-MANNING, C. G. and I. H. WITTEN (1997) "Identifying hierarchical structure in sequences: a linear-time algorithm," *Journal of Artificial Intelligence Research*, **7**(1), pp. 67–82.

[106] LI, X., X. WANG, and W. CHANG (2014) "CipherXRay: Exposing Cryptographic Operations and Transient Secrets from Monitored Binary Execution," *IEEE Transactions on Dependable and Secure Computing*, **11**(2).

[107] ZHAO, R., D. GU, J. LI, and Y. ZHANG (2013) "Automatic Detection and Analysis of Encrypted Messages in Malware," in *Proceedings of the 9th China International Conference on Information Security and Cryptology (INSCRYPT'13)*.

[108] OH, J. W. (2009) "Fight against 1-day exploits: Diffing Binaries vs Anti-diffing Binaries," in *2009 Black Hat USA*.

[109] WANG, P., S. WANG, J. MING, Y. JIANG, and D. WU (2016) "Translingual Obfuscation," in *Proceedings of the 1st IEEE European Symposium on Security and Privacy (Euro S&P'16)*.

[110] XU, D., J. MING, and D. WU (2016) "Generalized Dynamic Opaque Predicates: A New Control Flow Obfuscation Method," in *Proceedings of the 19th Information Security Conference (ISC'16)*.

[111] MING, J., Z. XIN, P. LAN, D. WU, P. LIU, and B. MAO (2015) "Replacement Attacks: Automatically Impeding Behavior-based Malware Specifications," in *Proceedings of the 13th International Conference on Applied Cryptography and Network Security (ACNS'15)*.

[112] ——— (2016) "Impeding Behavior-based Malware Analysis via Replacement Attacks to Malware Specifications," *Journal of Computer Virology and Hacking Techniques (In press)*.

[113] GAO, D., M. REITER, and D. SONG (2008) "BinHunt: Automatically finding semantic differences in binary programs," in *Poceedings of the 10th International Conference on Information and Communications Security (ICICS'08)*.

[114] NG, B. H. and A. PRAKASH (2013) "Exposé: Discovering Potential Binary Code Re-use," in *Proceedings of the 37th IEEE Annual Computer Software and Applications Conference (COMPSAC'13)*.

[115] MING, J., D. XU, and D. WU (2015) "Memoized Semantics-Based Binary Diffing with Application to Malware Lineage Inference." in *Proceedings of the 30th IFIP TC-11 SEC International Information Security and Privacy Conference (IFIP SEC'15)*.

[116] ——— (2016) "MalwareHunt: Semantics-Based Malware Diffing Speedup by Normalized Basic Block Memoization," *Journal of Computer Virology and Hacking Techniques (In press)*.

[117] SLOWINSKA, A. and H. BOS (2009) "Pointless Tainting?: Evaluating the Practicality of Pointer Tainting," in *Proceedings of the 4th ACM European Conference on Computer Systems (EuroSys'09)*.

[118] CAVALLARO, L., P. SAXENA, and R. SEKAR (2008) "On the Limits of Information Flow Techniques for Malware Analysis and Containment," in *Proceedings of the GI International Conference on Detection of Intrusions & Malware, and Vulnerability Assessment (DIMVA'08)*.

[119] WANG, S., P. WANG, and D. WU (2016) "Uroboros: Instrumenting Stripped Binaries with Static Reassembling," in *Proceedings of 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER'16)*.

Vita

Jiang Ming

Jiang Ming is currently a Ph.D. candidate in the College of Information Sciences and Technology of Pennsylvania State University, where he is a member of the Software Systems Security Research Lab. His research focuses on security, especially software security and malware analysis, including secure data flow analysis, software plagiarism detection, malicious binary code analysis, and software analysis for other security issues. He received the B.S. degree in Information Security from Wuhan University in 2006 and the M.Eng. degree in software Engineering from Peking University in 2009, respectively.