On Efficiency and Accuracy of Data Flow Tracking Systems

Kangkook Jee

Submitted in partial fulfillment of the

requirements for the degree

of Doctor of Philosophy

in the Graduate School of Arts and Sciences

COLUMBIA UNIVERSITY

2016

ABSTRACT

On Efficiency and Accuracy of Data Flow Tracking Systems

Kangkook Jee

Data Flow Tracking (DFT) is a technique broadly used in a variety of security applications such as attack detection, privacy leak detection, and policy enforcement. Although effective, DFT inherits the high overhead common to in-line monitors which subsequently hinders their adoption in production systems. Typically, the runtime overhead of DFT systems range from $3\times$ to $100\times$ when applied to pure binaries, and $1.5\times$ to $3\times$ when inserted during compilation. Many performance optimization approaches have been introduced to mitigate this problem by relaxing propagation policies under certain conditions but these typically introduce the issue of inaccurate taint tracking that leads to over-tainting or under-tainting.

Despite acknowledgement of these performance / accuracy trade-offs, the DFT literature consistently fails to provide insights about their implications. A core reason, we believe, is the lack of established methodologies to understand accuracy.

In this dissertation, we attempt to address both *efficiency* and *accuracy* issues. To this end, we begin with libdft, a DFT framework for COTS binaries running atop commodity OSes and we then introduce two major optimization approaches based on statically and dynamically analyzing program binaries.

The first optimization approach extracts DFT tracking logics and abstracts them using *TFA*. We then apply classic compiler optimizations to eliminate redundant tracking logic and minimize interference with the target program. As a result, the optimization can achieve $2\times$ speed-up over base-line performance measured for libdft. The second optimization approach decouples the tracking logic from execution to run them in parallel leveraging modern multi-core innovations. We apply his approach again applied to libdft where it can run *four times as fast*, while concurrently consuming *fewer* CPU cycles.

We then present a generic methodology and tool for measuring the accuracy of arbitrary DFT systems in the context of real applications. With a prototype implementation for the Android framework – TaintMark, we have discovered that TaintDroid's various performance optimizations lead to serious accuracy issues, and that certain optimizations should be removed to vastly improve accuracy at little performance cost. The TaintMark approach is inspired by blackbox differential testing principles to test for inaccuracies in DFTs, but it also addresses numerous practical challenges that arise when applying those principles to real, complex applications. We introduce the TaintMark methodology by using it to understand taint tracking accuracy trade-offs in TaintDroid, a well-known DFT system for Android.

While the aforementioned works focus on the efficiency and accuracy issues of DFT systems that dynamically track data flow, we also explore another design choice that statically tracks information flow by analyzing and instrumenting the application source code. We apply this approach to the different problem of integer error detection in order to reduce the number of false alarmings.

Table of Contents

List of Figures

List of Tables

Acknowledgments

First and foremost, I would like to thank my advisor Angelos D. Keromytis for giving me the opportunity and supporting me throughout my graduate studies. I honestly could not hope for a better advisor-student relationship.

My friend and close collaborator professor Gorgios Protokalidis deserves a big thank you. His guidance was an essential ingredient for the success of this work.

I would like to thank all the members of the NSL group, and I would like to thank everyone in the CS department that helped along the way.

Nothing would be possible without the support, dedication and love of Sunghee Choi. My last words are reserved for Inyoo and Sunyoo. You are endless sources for my joy, happiness, and inspiration.

Part I

Introduction and Background

Chapter 1

Introduction

1.1 Motivation

Dynamic data flow tracking (DFT), also known as Information Flow Tracking (IFT), is a well established technique that deals with the tagging and tracking of "interesting" data as they propagate during program execution. In the modern computing environment, DFT has gained a lot of attention both from the research community and the industry for its verified usefulness and variety of application domains. For instance, DFT has been used to protect software against exploits [Newsome and Song, 2005; Costa *et al.*, 2005], to analyze malware [Yin *et al.*, 2007; Lee *et al.*, 2013], to discover bugs [Nethercote, 2004; Chipounov *et al.*, 2011], to reverse engineer programs [Slowinska *et al.*, 2011], to control information flows [Zhu *et al.*, 2011], to audit privacy policies [Sen *et al.*, 2014], to tune software performance [Attariyan *et al.*, 2012], to uncover the structure of application-level data objects in operating systems [Spahn *et al.*, 2014], and to increase users' visibility into mobile data leakage from their applications [Enck *et al.*, 2010]. However, the wide adoption of this technology in production systems is hindered for the following reasons:

High overhead. As we have not yet seen explicit support for DFT in commodity hardware, the only practical means for applying these techniques are through various software-based implementations. Unfortunately, these implementations exhibit prohibitive performance overhead, that ranges from $3\times$ to $100\times$ when applied to pure binaries [Portokalidis *et al.*, 2006; Newsome and Song, 2005; Clause *et al.*, 2007], and $1.5\times$ to $3\times$ when inserted during compilation [Xu *et al.*, 2006; Lam and Chiueh, 2006]. The high overhead problem prevents the technology from being deployed to

performance sensitive domains. Even when performance is not an issue, the overhead can still be problematic: (a) if it changes the behavior of the application (e.g., when network connections timeout or when the analysis is no longer transparent), or (b) when computational cycles are scarce or CPU energy consumption needs to be kept to a minimum, as in mobile devices.

The absence of accuracy evaluations in DFT literature.

Although the high overhead issue is well recognized by the research community and a lot of effort was put in developing practical DFT systems with good performance and scalability, the issue of accurate tracking is often overlooked. Many of the performance improvement efforts can also lead to *imprecise tracking* (taints are propagated without any information flow) or *incomplete tracking* (taints are not propagated where information does flow). Despite acknowledgement of these accuracy/performance tradeo-ffs, the literature rarely provides any insight about these trade-offs. More broadly, a survey of nine papers proposing DFT frameworks with potential for inaccuracies reveals that accuracy trade-offs are rarely evaluated in the DFT literature [Enck *et al.*, 2010; Enck *et al.*, 2014; Tripp and Rubin, 2014; Kemerlis *et al.*, 2012; Dalton *et al.*, 2007; Portokalidis *et al.*, 2006; Qin *et al.*, 2006; Xu *et al.*, 2006].

1.2 Research Approach

In this dissertation, we introduce our approaches to address both limitations of DFT technology. First, we begin by implementing a practical DFT framework and then propose optimization approaches that would improve the runtime performance of the framework and subsequently make it become closer to being practical. Secondly, we propose a methodology to evaluate the accuracy of DFT systems and a prototype that implements this methodology. We used this prototype to evaluate the accuracy of TaintDroid, a DFT for the Android framework running real world applications. In the following paragraphs, we outline research requirements and challenges that we encountered from each component.

1.2.1 DFT Prototype

As a first step, we need a solid DFT framework. We prototype a DFT framework – libdft with the following design requirements in mind to address limitations of previous DFT proposals. The

DFT system should be concurrently *fast*, *reusable*, and *portable* by being applicable to commodity hardware and software.

libdft. In response to the above requirements, we implemented libdft [Kemerlis *et al.*, 2012] a DFT system that performs fine-grained (byte-level) data flow tracking operations to track explicit data flows (see Section 2.1.2). For the purpose of this prototype, libdft does not track implicit flow (information flow by control transfer) for two reasons – (a) it incurs too high overhead, and (b) it produces a significant number of false positive reports, indicating that it should be used only in certain use cases such as malware analysis.

1.2.2 DFT Optimizations

Previous binary-only DFT frameworks (including libdft) operate by individually instrumenting each program instruction that propagates data, adding one or more instructions that implement the data tracking logic. Higher-level semantics are not available and cannot be easily extracted, for example, it is not possible to automatically determine that a function copies data from one buffer to another, and directly generates the tracking code for that. This introduces a lot of redundant instructions and greatly reduces the proportion of "productive" instructions in the program. The problem is exacerbated in register-starved architectures like Intel's *x86*, where the program and data tracking logic compete for general purpose registers. This calls for an approach that would widen the scope of analysis by restoring higher-level program semantics (e.g., function or CFG) not limited to an instruction level analysis. Using this extended analysis scope, optimizations that aggressively eliminate redundant taint operations are rendered possible.

Taint Flow Algebra (TFA). TFA is an optimization approach to improve the performance of DFT frameworks by combining static and dynamic analysis. Our methodology separates program logic from taint tracking logic, extracting the semantics of the latter, and represents them using a TFA. We then apply multiple code optimization techniques to eliminate redundant tracking logic and minimize interference with the target program in a manner similar to an optimizing compiler. Albeit TFA can reduce the size of taint tracking logic and instrumentation frequency, it still instruments (optimized) tracking logic with the application execution logic subsequently having two logics to run from a single process (or thread) and compete for shared system resources such as address space, CPU registers, etc. This causes the underlying instrumentation framework (in our case PIN

DBI [Luk *et al.*, 2005]) to incur high scheduling overhead [1].

ShadowReplica In order to address the limitation of TFA approach, ShadowReplica introduces an efficient approach for accelerating DFT and other shadow memory-based analyses [Nethercote, 2004], by *decoupling* analysis from execution and utilizing spare CPU cores to run them in parallel. This approach enables us to run a heavyweight technique like dynamic taint analysis (DTA) *twice as fast* over TFA, while concurrently consuming *fewer* CPU cycles. The main motivation behind ShadowReplica has been to accelerate the DFT technique, but it can also host other types of analyses. We demonstrate this by also implementing a control-flow integrity (CFI) [Abadi *et al.*, 2005] tool over ShadowReplica.

1.2.3 Static IFT Framework for Integer Errors

Even after applying the optimizations, our fastest DFT prototype (ShadowReplica) still incurs high overhead making it unsuitable for certain application domains. This limitation is fundamental to our approach that dynamically instruments tracking logics into the binary programs using PIN DBI [Luk *et al.*, 2005]. As an alternative, we explore a different design direction that statically analyzes and instruments tracking at compile time and applying this approach to improve the accuracy of arithmetic error detection, focusing on reducing the number of false positives, i.e., developer-intended code constructs that violate language standards. Our prototype, IntFlow, uses information flow tracking to reason the severity of arithmetic errors by analyzing the information flows related to them.

Although we observe that the approach based on static IFT incurs less overhead , we can also confirms that this approach discloses more accuracy issues than the dynamic approach. For instance, owing to limitation of its static IFT component [Moore, 2013], IntFlow's inter-procedural dataflow analysis cannot be extended beyond a certain number of function calls. Point-to analysis, crucial in relating two variables and defining data flow, is another source of inaccuracy. IntFlow is therefore unable to detect every information flow that would lead to false reporting of integer errors requiring user interventions in some corner-cases. In other words, IntFlow approach can reduce the number

[1] We observed $\sim$ 1.6x slowdown when we run SPEC CPU2006 benchmark suite from PIN DBI with empty (`null`) analysis routine. This overhead reflects pure instrumentation cost and it tends to increase when having larger and more frequent analysis routines.

of false reports but not capable of detecting all incorrect reports.

1.2.4 Measuring the Accuracy of DFT frameworks

While we expect that the static IFT approach to have accuracy limitations [Aho *et al.*, 2006] , the accuracy aspect of dynamic DFT approach is less obvious. However, in the pursuit of efficiency, many DFT implementations choose to relax tracking policies in certain cases and these lead to accuracy problems in terms of false positives and false negatives. TaintDroid [Enck *et al.*, 2010], a DFT implementation for the Android framework, is widely acknowledged for its minimal runtime overhead which is approximately 14% over native execution, and which is achieved by relaxing the number of important tag propagation policies virtually implementing coarse grained tracking for certain system operations. For instance, system objects like arrays, file, and Android IPC binder objects, TaintDroid only assigns a single tag to represent the taint for all elements in an array, different fields of binder object, and all contents of a file object.

While the number of inaccuracy issues of TaintDroid are found and reported [Tang *et al.*, 2012; Spahn *et al.*, 2014], the research community still fails to provide useful insight about performance and accuracy tradeoffs or status quo. We survey a number of previous works [Enck *et al.*, 2010; Enck *et al.*, 2014; Tripp and Rubin, 2014; Kemerlis *et al.*, 2012; Dalton *et al.*, 2007; Portokalidis *et al.*, 2006; Qin *et al.*, 2006; Xu *et al.*, 2006] proposing DFT frameworks with potential for inaccuracies, but we failed to find works that systematically evaluate the accuracy of their DFT system. We believe that a core reason for the absence of accuracy evaluations in the DFT literature is the lack of an established methodology for measuring accuracy, as well as not having the tools to facilitate its adoption. Indeed, evaluating the accuracy is challenging, because such an evaluation requires ground truth for information flows. While ground truth can be established for simple test benchmarks, it is typically infeasible to infer the ground truth for real world applications. Increased visibility into the accuracy of DFT systems – as we believe is necessary for practical DFT designs – requires testing the DFT technologies on real applications, where true information flows are unknown.

To address this challenge, we developed *TaintMark*, a new methodology and tool for to understand the accuracy of DFT systems with real applications. TaintMark uses blackbox differential testing to find likely inaccuracies (incomplete or imprecise tracking) in a specified DFT system,

such as TaintDroid or any other DFTs.

1.3 Contributions

My thesis comprises in two statements,

- Optimizations can improve runtime performance of DFT systems without compromising propagation accuracy.
- We can measure the accuracy of DFT systems by co-relating inputs and outputs for a typical program operations.

In this work, we make the following contributions:

- We discuss the design and implementation of a fast and reusable DFT framework for commodity software.
- We demonstrate a methodology for segregating program logic from data tracking logic, and optimizing the latter separately using a combination of static and dynamic analysis techniques. Our approach is generic and can benefit most binary-only DFT approaches.
- We propose an approach to efficiently parallelize in-line analysis by implementing low-cost communication between the primary original process and the secondary analyzer process. This approach preserves the functionality of both the original program being monitored, and the analysis logic that would be otherwise applied in-line.
- We present an accurate arithmetic error detection approach that leverages static Information Flow Tracking (IFT) approach.
- We propose the first practical methodology and tool for understanding the accuracy of DFTs in complex real applications. With a prototype that implements the methodology, we present a detailed study of accuracy in the TaintDroid mobile DFT and its variants.
- Our implementations are open source and freely available thus they can be used to extend the DFT technology or to develop new tools.

1.4 Dissertation Roadmap

In this dissertation, we discuss how we implement and improve DFT systems in term of efficiency and accuracy.

Chapter 2 covers relevant background and related work that lead to this dissertation. In Chapter 3, we discuss the design and implementation of libdft, our DFT prototype. In Chapter III, we discuss two major optimization apporaches to improve DFT frameworks, namely Taint Flow Algebra (TFA) and ShadowReplica. In Chapter 6, we discuss the design and implementation of IntFlow, a tool for integer error reporting based on static IFT. In Chapter 7, we introduce TaintMark a methodology and framework to evaluate the accuracy of DFT systems. Finally, we conclude the dissertation from Chapter 8.

Chapter 2

Background and Related Work

This chapter intends to give an overview of the basic concepts required for the better understanding of the following chapters as well as provide a brief survey of the most relevant and influential research efforts in the related areas.

2.1 Background

2.1.1 In-line Monitoring

Program protection and profiling using inline monitors is a dynamic approach that executes specific analysis logics along with the application process. Instances of this technology includes DFT, memory integrity checking [Nethercote and Seward, 2007; Bruening and Zhao, 2011], control flow integrity [Abadi *et al.*, 2005], method counting, call graph profiling etc. Given that the technology can be implemented by interleaving analysis and monitoring logics into the program execution, we can choose to instrument either source code or program binary and therefore use different frameworks to implement inline monitors.

Source Code Instruementation. When instrumenting the source code, we can make use of compiler internal representations [Lattner and Adve, 2004] such as abstract syntax tree (AST) or intermediate representation (IR) or use source-to-source transformation [Cordy, 2006; Necula *et al.*, 2002] to generate new source code embedded with monitoring logics. This approach induces reasonable overhead of roughly $\times 2$ or less, but it is limited in coverage since COTS binaries (i.e., 3rd

```
1: unsigned char csum = 0;
2:
3: bcount = read(fd, data, 1024);
4: while(bcount-- > 0)
5:       csum ^= *data++;
6:
7:  write(fd, &csum, 1);
```

(a) Data flow dependency

```
1: int authorized = 0;
2:
3: bcount = read(fd, pass, 12);
4: MD5(pass, 12, phash);
5: if (strcmp(phash, stored_hash) == 0)
6:     authorized = 1;
7: return authorized;
```

(b) Control flow dependecy

Fig. 2.1: Examples of code with data dependencies.

party libraries) are not supported.

Binary Instrumentation. An alternative that overcomes this limitation is to leverage binary instrumentation based on either process-wide virtualization using dynamic binary instrumentation (DBI) [Luk *et al.*, 2005; Bruening, 2004; Nethercote, 2004] or system-wide virtualization [Bellard, 2005; Barham *et al.*, 2003]. However, it comes with an excessive amount of overhead which varies from $\times 5 \sim \times 100$ based on analyses and application domains.

Instrumentation with Hardware Support. Hardware assisted implementation [Dalton *et al.*, 2007; Chen *et al.*, 2008] is considered to implement inline monitors with minimal overhead usually less than 5% with full coverage of execution environment, however the we have not seen this being supported by major vendors in their commodity products yet.

Instrumentation with Language Runtimes. Programs written in high level languages (e.g., Java, python, C#) are firstly translated into bytecode (or intemediate representation) and then executed from their own runtime interpreters run as a process. This execution model benefits users in different ways at the expense of non-negligible amount of runtime performance penalty. Firstly, it supports protable execution of the program by having the language runtime inteface with underlying OS. Secondly, it provides controlled and isolated envrionment by not disclosing the complex details of system operations (i.e., memory management) to programs. Along with these benefits and flexibility, the task of instrumenting in-line monitor becomes easier from this model. We can either transform the bytecode or modify language runtime to execute in-line logics along with the program.

2.1.2 Data Flow Tracking

DFT has been a popular subject of research, primarily employed for enforcing safe information flow and identifying illegal data usage. In past work, it is frequently referred to as Information

Flow Tracking (IFT) [Suh *et al.*, 2004] or taint analysis. This work defines DFT as: "the process of *accurately* tracking the flow of *selected* data throughout the execution of a program or system". This process is characterized by three aspects, which we will attempt to clarify with the help of the code-snippets shown in Figure 2.1.

Data sources Data sources are program or memory locations, where data of interest enter the system, usually after the execution of a function or system call. Data coming from these sources are tagged and tracked. For instance, if we define files as a source, the `read` call in Figure 2.1 would result in tagging `data` and `pass`.

Data tracking During program execution, tagged data are tracked as they are copied and altered by program instructions. Consider code snippet (a) in Figure 2.1, where `data` has already been tagged in line 3. The while loop that follows calculates a simple checksum (XOR all the bytes in `data`) and stores the result in `csum`. In this case, there is a data flow dependency between `csum` and `data`, since the former directly depends on the latter. On the other hand, `authorized` in (b) is indirectly affected by the value of `phash`, which in turn depends on `pass`. This is frequently called a control flow dependency, and in this work, we do not consider cases of implicit data flow that are in accordance with previous work on the subject [Newsome and Song, 2005; Suh *et al.*, 2004]. Dytan made provisions for conditionally handling such control-flow dependencies, but concluded that, while useful in certain domains, they frequently lead to an explosion in the amount of tagged data and to incorrect data dependencies [Clause *et al.*, 2007]. Ongoing work seeks to address these issues [Kang *et al.*, 2011].

Data sinks Data sinks are also program or memory locations, where one can check for the presence of tagged data, usually for inspecting or enforcing data flow. For instance, tagged data may not be allowed in certain memory areas and function arguments. Consider again the code-snippet (a) in Figure 2.1, where in line 7 `csum` is written to a file. If files are defined as data sinks, the use of `write` with `csum` can trigger a user-defined action.

2.2 Related Work

2.2.1 DFT Implementations and Optimizations

DFT Implementations. Dytan [Clause *et al.*, 2007], Minemu [Bosman *et al.*, 2011], LIFT [Qin *et al.*, 2006], and are previously proposed dynamic DFT systems. Dytan is the most flexible tool, allowing users to customize its sources, sinks, and propagation policy, while it can also track data based on control-flow dependencies (see Figure 2.1). Albeit flexible, it incurs high performance penalties and it can also lead to taint explosion. That is, erroneously tracking large amounts of data, due to the imprecision of control-flow data dependencies [Slowinska and Bos, 2009]. In contrast, Minemu is the fastest tool, but it provides limited functionality. It uses an ad hoc emulator for the mere purpose of performing fast DTA, and cannot be configured for use in other domains without modifying the emulator itself. Moreover, it does not provide colored tags, nor it supports self-modifying code. More importantly, Minemu cannot be used "as-is" on 64-bit architectures, due to its shadow memory design and heavy reliance on SSE (XMM) registers.

TaintCheck [Newsome and Song, 2005] was one of the first tools to utilize DTA, for protecting binary-only software from buffer overflow and other types of memory corruption attacks, entirely in software. Eudaemon [Portokalidis and Bos, 2008] builds upon the QEMU user space emulator to allow one selectively apply taint analysis on a process (e.g., when potentially harmful actions are taken, or the system is idle). It incurs an overhead of approximately 9x, which can be alleviated on long-running applications by selectively enabling or disabling DFT.

Hardware implementations of DFT [Suh *et al.*, 2004; Dalton *et al.*, 2008; Crandall and Chong, 2004] have been proposed to evade the large penalties imposed by software-only implementations, but unfortunately they have had no appeal with hardware vendors. Implementations of DTA using virtual machines and emulators have been also proposed [Portokalidis *et al.*, 2006; Ho *et al.*, ; Chow *et al.*, 2004]. While these solutions have some practicality for malware analysis platforms and honeypots, they induce slowdowns that make them impractical on production systems.

HiStar [Zeldovich *et al.*, 2006] uses labels to tag and protect sensitive data. It is a new OS design, and its main focus is to protect the OS from components that start exhibiting malicious behavior after being compromised. HiStar's data tracking is more coarse-grained than our prototypes, and introduces major changes in the OS level.

DFT Optimization Approaches. DFT approaches that operate on binary-only software typically incur high slowdowns ranging from $3\times$ to $100\times$ on CPU-intensive workloads [Qin *et al.*, 2006; Chow *et al.*, 2004]. Much research focused on the overhead of DFT for systems using binary instrumentation and virtualization, which suffer the largest overheads. Important paths explored by previous research include:

- **Additional resources** Data tracking is decoupled from program execution and is offloaded to one or more CPUs [Nightingale *et al.*, 2008; Ruwase *et al.*, 2008] or even to remote hosts [**?**; Portokalidis *et al.*, 2010]. These techniques can also parallelize the tracking itself and scale to multiple cores.

- **Static analysis** The application is statically analyzed before inserting the tracking code to avoid injecting redundant tracking instructions [Lam and Chiueh, 2006]. The speed-up can be considerable when narrowing down the data tracking to particular elements of a program [Saxena *et al.*, 2008], while the analysis can also be user assisted [Zhu *et al.*, 2011].

- **Intermittent tracking** Performance is improved by dynamically deactivating data tracking under certain conditions (e.g., when the program does not operate on tagged data [Qin *et al.*, 2006; Ho *et al.*,]). Data tracking can also be applied on-demand based on CPU usage, or due to manual activation [Portokalidis and Bos, 2008]. Tracking only parts of an application also reduces overhead [Kim and Keromytis, 2009].

Many of these approaches improve performance, but are not always costless. For instance, approaches utilizing additional hardware resources do not always scale (i.e., adding CPUs does not further improve performance). Also, static analysis methods frequently require access to source code, and oftentimes sacrifice functionality and/or accuracy. Intermittent tracking offers significant performance improvements on certain favorable scenarios, but the cost of activating and deactivating it can actually increase overhead on unfavorable ones.

The optimizations we describe in this disseration are based on statically and dynamically analyzing program binaries. More importantly though, they can be combined with some of the approaches listed above to offer additive performance benefits. For example, our approach can optimize the tracking logic that runs on itself on spare core or remote host. It can also be combined with other

static analysis approaches. For instance, Saxena et al. [Saxena *et al.*, 2008] use static analysis to recover the high-level semantics of a program (e.g., local variables, stack conventions, etc). Their approach is not applicable to all binaries, but it is orthogonal to ours. Finally, our methodology can also benefit intermittent tracking approaches to accelerate the tracking logic when it is activated.

The idea of decoupling dynamic program analyses from execution, to run them in parallel, has been studied in past in various contexts [Wallace and Hazelwood, 2007; Chow *et al.*, 2008; Nightingale *et al.*, 2008; Zhao *et al.*, 2008; Ha *et al.*, 2009; Portokalidis *et al.*, 2010; Chen and Chen, 2013]. Aftersight [Chow *et al.*, 2008],ReEmu [Chen and Chen, 2013], and Paranoid Android [Portokalidis *et al.*, 2010] leverage record and replay for recording execution and replaying it, along with the analysis, on a remote host or a different CPU (replica). They are mostly geared toward off-line analyses and can greatly reduce the overhead imposed on the application. However, the speed of the analysis itself is not improved, since execution needs to be replayed and augmented with the analysis code on the replica.

2.2.2 DFT Accuracy Study

Input-Output Influence. Examining the output of a system using information obtained from the input is by no means a new concept. Millen [Millen, 1987] and Clark et al [Clark *et al.*, 2001; Clark *et al.*, 2005] use the concept of entropy introduced by Shannon information theory to quantitatively measure the amount of information flow between input and output. Newsome et al [Newsome *et al.*, 2009] take a similar approach but use channel capacity instead of entropy to relate the input with the output. Clarkson et al [Clarkson *et al.*, 2005] propose a model that takes into account the belief of the attacker in every state of execution to measure how the input affects the output. Finally, BayesDroid [Tripp and Rubin, 2014] is a system that employs a naive Bayes classifier to make more sophisticated decisions on which reported leaks of DFT systems are false positives. It comes in two flavors regarding the tag propagation, one that employs TaintDroid and another that uses a "brute-force" approach. McCamant et al [McCamant and Ernst, 2008] describe a technique that transforms the problem of privacy leaking to the classic maximum-flow/minimum-cut problem. They construct circuit-like graphs based on an execution and compute the maximum-flow that represents the information-flow of this run while the produced minimum-cut represents a set of values that connects the secret input to the output.

Black-Box Input-Output Correlation. Two closely related works are Privacy Oracle [Jung *et al.*, 2008] and XRay [Lecuyer *et al.*, 2014], both of which leverage differential fuzz testing to identify information flows in black-box programs. The Privacy Oracle identifies flows on x86 systems, while XRay identifies them within and across live Web services. Our work is different from theirs in the following aspects: (1) we aim to understand accuracy (both false positives and false negatives) in DFT systems by blackbox differential testing, where we solve challenges stemmed from system non-determinism, (2) we present a debugging component that can help developers pinpoint the root causes of accuracy bugs, and (3) we present the first in-depth analysis of accuracy of TaintDroid and its variants.

Part II

DFT Implmentations

Chapter 3

Fast and Efficient DFT System for COTS Binaries

In this section, we discuss how we build libdft our foundation DFT prototype. We also explore number of design and implementation choices to consider to build DFT system that satisfies our three design goals – DFT system which is *fast*, *reusable*, and appliable to *commidity software and hardwares*.

3.1 libdft Overview

libdft demonstrates that a practical dynamic DFT implementation that addresses following features. It is concurrently *fast*, *reusable*, and *applicable to commodity hardware and software*. libdft is a meta-tool in the form of a shared library that implements dynamic DFT using Intel's Pin dynamic binary instrumentation framework [Luk *et al.*, 2005]. libdft's performance is comparable or better than previous work, incurring slowdowns that range between 1.14x and 6.03x for command-line utilities, while it can also run large server applications like Apache and MySQL with an overhead ranging between 1.25x and 4.83x. In addition, it is versatile and reusable by providing an extensive API that can be used to implement DFT-powered tools. Finally, it runs on commodity systems. Our current implementation works with x86 binaries on Linux, while we plan to extend it to run on 64-bit architectures and the Windows operating system (OS). libdft introduces an efficient, 64-bit capable, shadow memory, which represented one of the most serious limitations of earlier works, as

flat shadow memory structures imposed unmanageable memory space overheads on 64-bit systems, and dynamically managed structures introduce high performance penalties. More importantly, libdft supports multiprocess and multithreaded applications, by trading off memory for assurance against race conditions, and it does not require modifications to programs or the underlying OS.

The contributions of libdft can be summarized as follows:

- We discuss the design and implementation of a fast and reusable shared DFT library for commodity software. Specifically, we investigate and identify the underlying reasons responsible for the performance degradation incurred by previous DFT tools and present a design that minimizes it.

- We evaluate the performance of libdft using real applications that include complex and large software such as the Apache and MySQL servers, and the Firefox web browser. libdft achieves performance similar, or better, than previous work, while being applicable to a broader set of software.

- We present the development of a libdft-powered tool, namely libdft-DTA, to demonstrate the reusability of libdft as well as its capabilities. libdft-DTA performs dynamic taint analysis (DTA) to detect zero-day attacks similarly to TaintCheck [Newsome and Song, 2005], Eudaemon [Portokalidis and Bos, 2008], and LIFT [Qin *et al.*, 2006]. We show that our versatile API can be used for developing an otherwise complex tool in approximately 450 lines of C++ code.

- Our implementation is freely available as a shared library and can be used for developing tools that transparently (i.e., without requiring any change on applications or the underlying OS) make use of DFT services. Developers can use the API provided to easily define data of interest, and then capture their use at arbitrary points.

3.2 libdft Design

We designed libdft for use with the Pin DBI framework to facilitate the creation of Pintools that employ dynamic DFT. libdft is also implemented as a shared library, which can be used by Pintools to transparently apply fine-grained DFT on binaries running over Pin. More importantly, it provides

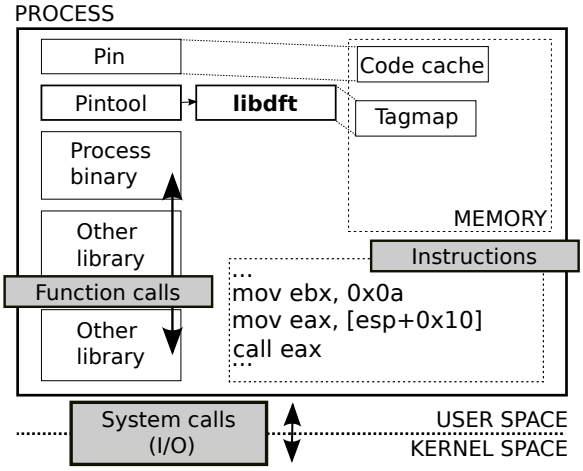


Fig. 3.1: Process image of a binary running under libdft. The highlighted boxes describe possible data sources and sinks that can be used with libdft.

its own API that enables tool authors to easily customize libdft by specifying data sources and sinks, or even modify the tag propagation policy.

When a user attaches to an already running process, or launches a new one using a libdft-enabled Pintool, the injector first injects Pin's runtime and then passes control to the tool. There are three types of locations that a libdft-enabled tool can use as a data source or sink: *program instructions*, *function calls*, and *system calls*. It can "tap" these locations by installing callbacks that get invoked when a certain instruction is encountered, or when a certain function or system call is made. These user-defined callbacks drive the DFT process by tagging or un-tagging data, and monitoring or enforcing data flow. Figure 3.1 sketches the memory image of a process running under a libdft-enabled Pintool. The highlighted boxes mark the locations where the tool author can install callbacks. For instance, the user can tag the contents of the buffer returned by the `read` system call and check whether the operands of indirect `call` instructions are tagged (e.g., the `eax` register in Figure 3.1).

3.2.1 Data Tags

libdft stores data tags in a tagmap (i.e., shadow memory), which contains a process-wide data structure (shadow memory) for holding the tags of data stored in memory and a thread-specific structure that keeps tags for data residing in CPU registers. he format of the tags stored in the tagmap is

determined by mainly two factors: (a) the granularity of the tagging, and (b) the size of the tags.

Tagging granularity In principle, we could tag data units as small as a single bit, or as larger contiguous chunks of memory. The former enables us to perform very fine-grained and accurate DFT, while using larger granularity means the data tracking will be coarser and more error prone. For instance, with page-level granularity, moving a single byte (tagged) into an untagged location will result into tagging the whole page that contains the destination, thus "polluting" adjacent data. However, choosing extremely fine-grained tagging comes at a significant cost, as more memory space is needed for storing the tags (e.g., using bit-level tagging, 8 tags are necessary for a single byte and 32 for a 32-bit register). More importantly, the tag propagation logic becomes complicated, since data dependencies are also more intricate (e.g., consider adding two 32-bit numbers that only have some of their bits tagged). libdft uses byte-level tagging granularity, since a byte is the smallest addressable chunk of memory in most architectures. Our choice allows us to offer sufficiently fine-grained tracking for most practical purposes and we believe that it strikes a balance between usability and performance [Portokalidis *et al.*, 2006].

Tag size Orthogonally to tagging granularity, larger tags are more versatile as they allow for different types of data to be tagged uniquely (e.g., each byte could be tagged using a unique 32-bit number). Unfortunately, larger tags require complex propagation logic and more storage space. libdft offers two different tag sizes: (a) byte tags for associating up to 8 distinct values or *colors* to each tagged byte (every bit represents a different tag class), and (b) single-bit tags (i.e., data are either tagged or not). The first allows for more sophisticated tracking and analysis tools, while the second enables tools that only need binary tags for conserving memory.

3.2.2 Tag Propagation

Tag propagation is accomplished using Pin's API to both instrument and analyze the target process. In Pin's terms, instrumentation refers to the task of inspecting the binary instructions of a program for determining what analysis routines should be inserted where. For instance, libdft inspects every program instruction that (loosely stated) moves or combines data to determine data dependencies. On the other hand, analysis refers to the actual routines, or code, being retrofitted to execute before, after, or instead of the original code. In our case, we inject analysis code implementing the tag propagation logic, based on the data dependencies observed during instrumentation.

The original code and libdft's analysis routines are translated by Pin's just-in-time (JIT) compiler for generating the code that will actually run. This occurs immediately before executing a code sequence for the first time, and the result is placed in a code cache (also depicted in Figure 3.1), so as to avoid repeating this process for the same code sequence. Our injected code executes before application instructions, tracking data as they are copied between registers, and between registers and memory, thus achieving fine-grained DFT. Pin's VM ensures that the target process runs entirely from within the code cache by interpreting all instructions that cannot be executed safely otherwise (e.g., indirect branches). Moreover, a series of optimizations such as trace linking and register re-allocation are applied for improving performance [Luk *et al.*, 2005].

Finally, libdft allows tools to modify the default tag propagation policy, by registering their own instrumentation callbacks via its API, for instructions of interest. This way tool authors can tailor the data tagging according to their needs, and cancel tag propagation in certain cases or track otherwise unhandled instructions.

3.2.3 Challenges for Fast Dynamic DFT

To keep libdft's overhead low, we carefully examined how DBI frameworks, such as Pin, work for identifying the development practices that should be avoided. Pin's overhead primarily depends on the size of the analysis code injected, but it can frequently be higher than anticipated due to the structure of the code itself. Specifically, the registers provided by the underlying architecture will be used to execute both application code, as well as code that implements the DFT logic, thus forcing the DBI framework to spill registers (i.e., save their contents to memory and later restore them), whenever an analysis routine needs to utilize registers already allocated. Therefore, the more complex the code, the more registers have to be spilled.

Additionally, certain types of instructions must be avoided due to certain side-effects. For instance, spilling the `EFLAGS` register in the x86 architecture is expensive in terms of processing cycles, and is performed by specialized instructions (`PUSHF, PUSHFD`). As a result, including instructions in analysis code that modify this register should be done sparingly. More importantly, test-and-branch operations have to be avoided altogether, since they result into non-inlined code. In particular, whenever a branch instruction is included in the DFT code, Pin's JIT engine will emit a function call to the corresponding analysis routine, rather than inline the code of the routine along

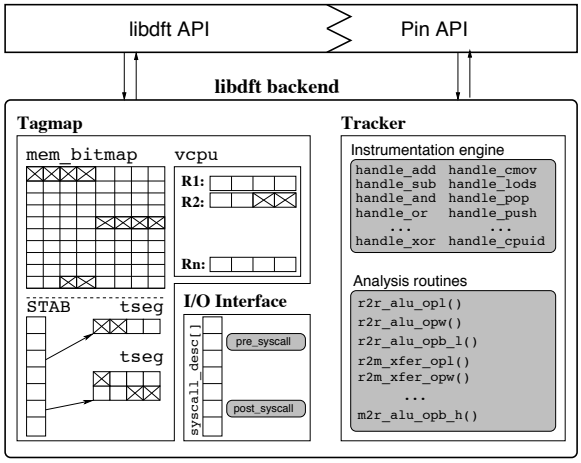


Fig. 3.2: The architecture of libdft. The shaded components of I/O interface and tracker illustrate the instrumentation and analysis code that implements the DFT logic, whereas the x-marked regions on the tagmap indicate tagged bytes.

with the instructions of the application. Imposing such limitations on the implementation of any dynamic DFT tool is a challenge. Our implementation takes into consideration these issues, in conjunction with Pin, to achieve good performance.

The design of libdft provides the foundation for a framework that satisfies three properties listed in Section 3.1. By taking into consideration the limitations discussed above, we achieve low overhead. Moreover, the extensive API of libdft makes it reusable, as it enables users to customize it for use in various domains, such as security, privacy, program analysis, and debugging. Finally, the last property is satisfied through the use of a mature, rather than an experimental and feature-limited, DBI platform for providing the apparatus to realize DFT for a variety of popular systems (e.g., x86 and x86-64 Linux and Windows OSs).

3.3 libdft Implementation

Our prototype works with unmodified multiprocess and multithreaded applications running on 32-bit x86 CPUs over Linux, the main components of libdft are illustrated in Figure 3.2.

3.3.1 Shadow Memory (Tagmap)

The implementation of the *shadow memory (tagmap)* plays a crucial role in the overall performance of libdft, since the injected DFT logic constantly operates on data tags.

3.3.1.1 Register Tags

We store the tags for all 8 general purpose registers (GPRs) of the x86 architecture in the `vcpu` structure, which is part of the tagmap (see Figure 3.2). Note that we tag and track only the registers that can be directly used by applications (like the GPRs). Registers such as the instruction pointer (`EIP`), `EFLAGS`, and segment registers, are not tagged or traced. In our definition of DFT and implementation of libdft, we only track direct data flow dependencies, so it is safe to ignore `EFLAGS`. However, instructions that are executed conditionally, based on the contents of `EFLAGS`, are handled appropriately (e.g., `CMOVcc`, `SETcc`). Moreover, floating point registers (`FPU`), as well as SSE registers (`XMM`, `MMX`), are currently ignored for the sake of simplicity. To support these registers in the future, we only need to enlarge `vcpu`.

The tagmap holds multiple `vcpu` structures, one for every thread of execution. Specifically, libdft captures the thread creation and thread termination events of an application and dynamically manages the number of `vcpu` structures. We locate the appropriate structure for each thread using its virtual id (i.e., an incremental value starting from zero) that is assigned by Pin to every thread. In the case of bit-sized tags, we use one byte to hold the four 1-bit tags needed for every 32-bit GPR, so that the space overhead of `vcpu` is 8 bytes for each thread. Similarly, in case of byte-sized tags, we need 4 bytes for every 32-bit GPR, and hence the space overhead of `vcpu` becomes 32 bytes per thread.

3.3.1.2 Memory Tags

Bit-sized tags When libdft is configured to use bit-sized tags, it stores memory tags in a flat, fixed-size structure (see `mem_bitmap` in Figure 3.2) that holds one bit for each byte of process addressable memory. The total size of the virtual address space in x86 systems is 4GB (2^{32}), however the OS reserves part of that space for itself (i.e., the kernel). The amount of space reserved for the kernel depends on the OS, with Linux usually adopting a 3G/1G memory split that leaves 3GB of

address space for processes. In this case, we require 384MB to be contiguously reserved for the tagmap.

The memory tags of address $vaddr$ can be obtained as follows: `tval = mem_bitmap[vaddr >> 3] & (MASK << (vaddr & 0x7))`. Specifically, we use the 29 most significant bits (MSBs) of $vaddr$, as byte index in `mem_bitmap`, for selecting a byte that contains the tags of $vaddr$. Then, we treat the 3 least significant bits of $vaddr$ as bit offset within the previously acquired byte, and by setting `MASK` to 0x1 we obtain the tag bit for a single byte. Similarly, if `MASK` is 0x3, or 0xF, we obtain the tag bits for a word, or double word, respectively. The address space overhead imposed by `mem_bitmap` is 12.5%. Using a fixed-size structure instead of a dynamically managed one (e.g., a page table-like one) allows us to avoid the penalties involved with managing and accessing it. Note that while on 32-bit systems the size of the tagmap is reasonable, flat bitmaps are not practical on 64-bit architectures. For instance, in x86-64 a flat bitmap would require 32TB.

Byte-sized tags When libdft is using byte-sized tags, it stores them in dynamically allocated tagmap segments (see `tseg` in Figure 3.2). Every time the application gets a new chunk of memory implicitly by performing an image load (e.g., when loading a dynamic shared object), or explicitly by invoking a system call like `mmap`, `mremap`, `brk`, and `shmat`, libdft intercepts the event and allocates an equally sized contiguous memory region. For instance, if the application requests an anonymous mapping of 1MB using `mmap`, libdft will "shadow" the allocated region with a tagmap segment of 1MB, for storing the byte tags of the `mmap`-ed memory. More importantly, tagmap segments that correspond to shared memory chunks are also shared. Hence, two processes running under libdft can effectively share shadow memory. To the best of our knowledge, we are the first to implement such a tag sharing scheme.

We obtain information about memory areas mapped at load time, or before libdft was attached on the application, through the `proc` pseudo-filesystem (`/proc/<pid>/maps`). This way we acquire the location of the stack and other kernel-mapped memory objects, such as the vDSO and vsyscall pages, and allocate the respective tagmap segments accordingly. In order to deal with the implicit expansion of the stack, libdft pre-allocates a tagmap segment to cover the stack as if it expands to its maximum value, which can be obtained via `getrlimit(RLIMIT_STACK)`. However, the same is not necessary for thread stacks, since they are allocated explicitly using `mmap`.

During initialization, libdft allocates a segment translation table (`STAB`) for mapping virtual

addresses to their corresponding bytes in tagmap segments. Since memory is given to processes in blocks that are multiples of page size, `STAB` entries correspond to page size areas. For each page, `STAB` stores an addend value, which is effectively a 32-bit offset that needs to be added to all memory addresses inside that page, for retrieving the respective tag metadata. Assuming again a 3G/1G memory split and 4KB pages, `STAB` requires 3MB (i.e., one entry for each 4KB in the range 0x00000000 – 0xBFFFFFFF). Whenever we allocate or free a tagmap segment, we update the `STAB` structure accordingly. Moreover, we ensure that segments that match with adjacent memory pages are also adjacent. This not only allows dealing with memory accesses crossing segment boundaries (e.g., unaligned multi-byte accesses that span two pages are valid in x86), but also enables us to use a simple operation to retrieve the tag for any memory address: `taddr = vaddr + STAB[vaddr >> 12]`. The 20 MSBs of $vaddr$ are used as index in `STAB` to get an addend value, which in turn added to $vaddr$ itself for obtaining the respective tag bytes.

Byte-sized tags let us tag data using 8 different colors, but incur a higher per-byte memory overhead. Additionally, dynamically managing the respective tagmap segments also introduces overhead. However, we proactively allocate tagmap segments whenever the application maps new memory, instead of lazily waiting until a tagmap segment is used, to avoid the penalties involved with using branching instructions in analysis routines.

3.3.2 Code Instrumentation and Analysis

The *tracker* is the core component of libdft that is in charge of instrumenting a program to retrofit the DFT logic. It consists of two parts, shown in Figure 3.2.

3.3.2.1 Instrumentation Engine

The instrumentation engine is responsible for inspecting program instructions to determine the analysis routine(s) that should be injected for each case. We use Pin's instrumentation API to inspect every instruction before it is translated by the JIT compiler. We first resolve the instruction type (e.g., arithmetic, move, logic), and then we analyze its operands for determining their category (i.e., register, memory address, or immediate) and length (byte, word, double word). After gathering this information, we use Pin to insert the appropriate analysis routine before each instruction.

The instrumentation code is invoked once for every sequence of instructions, and the result (i.e.,

the original code and analysis routines) is placed into Pin's code cache. We exploit code caching, by handling the x86 ISA complexity during the instrumentation phase, and keeping the analysis routines compact and fast. Specifically, we move the elaborate logic of discovering data dependencies and handling each variant of the same dependency category into the instrumentation phase. This allows us to aggressively optimize the propagation code by injecting compact, category-specific, and fast code snippets before each instruction. Due to the complexity and inherent redundancy of the ISA, our instrumentation engine consists of ∼3000 C++ lines of code (LOC).

3.3.2.2 Analysis Routines

The analysis routines contain the code that actually implements the DFT logic for each instruction. They are injected by the instrumentation engine before every instruction to assert, clear, and propagate tags, and unlike instrumentation code they execute more frequently (i.e., the analysis code injected for a specific instruction, executes every time the instruction executes).

Carefully implementing these analysis routines is paramount for achieving good performance. For instance, while Pin tries to inline analysis code into the application's code, the use of branch instructions will cause it to insert a function call to the respective routines instead (recall that function calls require extra processing cycles). The same also stands for overly large analysis routines. *Interestingly, we observed that the number of instructions, excluding all types of jumps, which Pin can inline is ∼20.*

For these reasons, we introduce two guidelines for the development of efficient tag propagation code: (i) tag propagation should be branch-less, and (ii) tagmap updates should be performed with a single assignment. Both of them serve the purpose of aiding the JIT process to inline the injected code and minimize register spilling. Our analysis routines are made up of ∼2500 C LOC, and include only arithmetical, logical, and memory operations. Moreover, we force Pin to use the *fastcall* x86 calling convention, for making the DFT code faster and smaller (i.e., the compiler will avoid emitting push, pop, or stack-based parameter loading instructions).

Tracking code can be classified in the following categories based on the corresponding instruction type (the numbers in parentheses indicate the total number of analysis routines we implemented for each class, which are necessary for capturing the semantics of different operand types and sizes):

- *ALU* (21): analysis routines for the most common x86 instructions that typically have 2 or

3 operands, such as `ADD`, `SUB`, `AND`, `XOR`, `DIV`, `IMUL`, and so forth. For such instructions, we take the *union* of the source and destination operand tags, and we store the result in the respective tags of the destination operand(s). Immediates are always considered untagged.

- *XFER* (20): this class includes data transfers from a register to another register (*r2r*), from a register to a memory location and vice-versa (*r2m* and *m2r*), as well as from one memory address to another (*m2m*). For this type of instructions the source operand tags are copied to the destination operand tags, and again, immediates are always considered untagged.

- *CLR* (6): certain fairly complex instructions always result in their operands being untagged. Examples of such instructions include `CPUID`, `SETxx`, etc Similarly, x86 *idioms* used for "zeroing" a register (e.g.,`xor eax, eax` and `sub eax, eax`), also result in untagging their operands.

- *SPECIAL* (45): this class includes analysis routines for x86 instructions that cannot be handled effectively with the aforementioned primitives, such as `XCHG`, `CMPXCHG`, `XADD`, `LEA`, `MOVSX`, `MOVZX`, and `CWDE`. For instance, although `XADD` can be handled by instrumenting the instruction twice with the respective XFER routines (for exchanging the tag values of the operands), and once with the ALU routine that handles `ADD`, the code size expansion would be prohibitive. Thus, we choose to implement an optimized analysis routine, for minimizing the injected code and inflicting less pressure on the code cache. libdft has one special handler for each quirky x86 instruction.

- *FPU, MMX, SSE*: these are ignored by default, unless their result is stored into one of the GPRs, or to a memory location. In such cases, the destination is untagged.

Figures 3.3(a) and 3.3(b) show excerpts from different types of analysis routines in the case of bit- and byte-sized tags, respectively. The `VIRT2BYTE` macro is used for getting the byte offset of a specific address in `mem_bitmap` (it performs a bitwise right shift by 3), whereas `VIRT2BIT` gives the bit offset within the previously acquired byte. `VIRT2STAB` is a macro for obtaining the `STAB` index given a virtual address (it performs a bitwise right shift by 12). Code snippets labeled as `alu` correspond to routines that instrument 2 operand instructions belonging to the ALU category. On the contrary, `xfer` indicates propagation code for instructions of the XFER category. The operand

```
--------------------[r2r_alu_opb_l]--------------------
threads_ctx[tid].vcpu.gpr[dst] |=
    threads_ctx[tid].vcpu.gpr[src] & VCPU_MASK8;
--------------------[r2m_alu_opw]--------------------
*((uint16_t *)(mem_bitmap + VIRT2BYTE(dst))) |=
    (threads_ctx[tid].vcpu.gpr[src] & VCPU_MASK16) <<
        VIRT2BIT(dst);
--------------------[m2r_alu_opl]--------------------
threads_ctx[tid].vcpu.gpr[dst] |=
    (*((uint16_t *)(mem_bitmap + VIRT2BYTE(src))) >>
--------------------[r2r_xfer_opb_l]-------------------
threads_ctx[tid].vcpu.gpr[dst] =
    (threads_ctx[tid].vcpu.gpr[dst] & ~VCPU_MASK8) |
    (threads_ctx[tid].vcpu.gpr[src] & VCPU_MASK8);
--------------------[r2m_xfer_opw]--------------------
*((uint16_t *)(mem_bitmap + VIRT2BYTE(dst))) =
    (*((uint16_t *)(mem_bitmap + VIRT2BYTE(dst))) &
        ~(WORD_MASK << VIRT2BIT(dst))) |
    ((uint16_t)(threads_ctx[tid].vcpu.gpr[src] &
        VCPU_MASK16) << VIRT2BIT(dst));
--------------------[m2r_xfer_opl]--------------------
threads_ctx[tid].vcpu.gpr[dst] =
    (*((uint16_t *)(mem_bitmap + VIRT2BYTE(src))) >>
        VIRT2BIT(src)) & VCPU_MASK32;
```

(a) Tag propagation code for bit-sized tags

```
--------------------[r2r_alu_opb_l]--------------------
*((uint8_t *)&threads_ctx[tid].vcpu.gpr[dst]) |=
    *((uint8_t *)&threads_ctx[tid].vcpu.gpr[src]);

--------------------[r2m_alu_opw]--------------------
*((uint16_t *)(dst + STAB[VIRT2STAB(dst)])) |=
    *((uint16_t *)&threads_ctx[tid].vcpu.gpr[src]);

--------------------[m2r_alu_opl]--------------------
threads_ctx[tid].vcpu.gpr[dst] |=
    *((uint32_t *)(src + STAB[VIRT2STAB(src)]));


--------------------[r2r_xfer_opb_l]-------------------
*((uint8_t *)&threads_ctx[tid].vcpu.gpr[dst]) =
    *((uint8_t *)&threads_ctx[tid].vcpu.gpr[src]);

--------------------[r2m_xfer_opw]--------------------
*((uint16_t *)&threads_ctx[tid].vcpu.gpr[dst]) =
    *((uint16_t *)&threads_ctx[tid].vcpu.gpr[src]);

--------------------[m2r_xfer_opl]--------------------
threads_ctx[tid].vcpu.gpr[dst] =
    *((uint32_t *)(src + STAB[VIRT2STAB(src)]));
```

(b) Analysis routine for byte-sized tags

Fig. 3.3: *(a)* Tag propagation code for various analysis routines when libdft is using bit-sized tags. *(b)* Code snippets for different analysis routines when libdft is using byte-sized tags.

size (i.e., 8-, 16-, 32-bit) is designated by the `op{b, w, l}` label suffix, while the `r2r`, `r2m`, and `m2r` prefix is used for specifying the operand type (register vs. memory).

In the case of byte-sized tags, achieving single assignment tagmap updates and branch-less tag propagation is relatively easy, due to the design of our shadow memory. Specifically, if both operands are registers, then we merely need to perform a copy or a bitwise OR operation, of the appropriate size, between the respective GPRs in the `vcpu` structure of the current thread. On the other hand, if one of the operands is a memory location, the effective address (i.e.,`src` or `dst` depending on the instrumented instruction) goes through `STAB` for getting the addend value that should be added to the address itself, in order to address the tag bytes from the corresponding tagmap segment (see Section 3.3.1.2). The final propagation is performed similarly to the previous case.

Note that when libdft is configured to use bit-sized tags, analysis routines tend to be larger and more elaborate. This is due to the pedantic bit operations that are required, since we cannot simply "move" separate bits between different tagmap locations. Hence, to avoid using branch instructions or multiple assignment statements, we resort in bit masks and bitwise operations.

3.3.3 I/O Interface

The *I/O Interface* is the component of libdft that handles the exchange of data between the kernel and the process through system calls. In particular, it consists of two small pieces of instrumentation code, namely the `pre_syscall` and `post_syscall` stubs, and a table of system call meta-information (`syscall_desc[]`), as illustrated in Figure 3.2. The `syscall_desc` table holds specific libdft-related information for all the 344 system calls of the Linux kernel (up to v2.6.39). For instance, it stores user-registered callbacks (for using a system call as a data source or sink), descriptors for the arguments of the call, or whether the system call writes data to user space memory. When a system call is made by the application, the stubs are called upon entry and exit. If the user has registered a callback function for a specific system call (either for entering or exiting), it is invoked. Otherwise, the default behavior of the `post_syscall` stub is to untag the data being written/returned by the system call. The advantages of this approach are twofold. First, we enable the tool writer to hook specific I/O channels (e.g., network I/O streams) and perform selective tagging. This way, the developer can customize libdft by using system calls as data sources and sinks. Second, we eliminate tag leaks by taking into consideration that some system calls write specific data to user-provided buffers. For example, consider `gettimeofday()` that upon each call overwrites user space memory that corresponds to one, or two, `struct timeval` data structures. Such system calls always result in sanitizing (i.e., untagging) the data being returned, unless the tool writer has installed a callback that selectively tags the returned data. Finally, hooking a function call is straightforward, and can be performed directly using libdft's and Pin's API.

3.3.4 Memory Protection

Dynamic DFT is frequently used to analyze malware or enforce security, and in that context it is desirable to guarantee the integrity of libdft by protecting its memory similarly to a sandbox [Ford and Cox, 2008]. As shown in Figure 3.1, the same address space is used by the application, Pin that allocates memory to store its code cache, and libdft that allocates the tagmap. Since all of the above reside inside the same space, the tracked program could accidentally or intentionally corrupt Pin or libdft.

Our solution to the problem is inspired by the scheme proposed by Xu et al. [Xu *et al.*, 2006], and relies on the premise that application code cannot execute natively without first being instrumented

and analyzed by libdft. Since we instrument all memory accesses, an instruction that tries to write at memory address $vaddr$ will be instrumented with the corresponding DFT logic, in order to assert or clear the respective tag(s) in tagmap. Therefore, by restricting access to specific blocks on the tagmap, we can prevent application code from accessing certain memory regions. In the case of bit-sized tags, we first enforce Pin to only use memory in a specific memory range (e.g., in the lower 512MB, 0x00000000 – 0x20000000). Then, we proceed to protect the bits of `mem_bitmap` that correspond to that range (e.g., the first 64MB). Whenever application code tries to access the lower 512MB of its virtual address space, the DFT analysis routines will access the protected blocks of the tagmap, leading to a memory violation error. However, note that the protected memory region cannot be arbitrary, and depends on the architecture's page size. Assuming a 4KB page size and 1-bit tags, both the beginning and ending of the protected memory range need to be aligned to 32K (8 bits per byte $\times$ 4KB page).

In the case of byte-sized tags, we allocate a *guard* page using `mmap` and set its access bits to `PROT_NONE`, effectively disallowing any access to it. During initialization, we set all `STAB` entries that correspond to unallocated memory pages to point to the guard page. From the application's perspective, Pin's and libdft's memory is always considered unallocated space. Hence, any access to these addresses will result in DFT logic operating on the guard page, leading again to a memory violation error. Note that by using libdft's API tool writers can register callback handlers for dealing with such violation errors.

3.4 libdft Evaluation

We evaluated libdft using a variety of software including a web and database (DB) server, command-line and network utilities, a web browser, and the SPEC CPU2000 suite. Our aim is to quantify the performance of libdft, and establish a set of bounds for the various types of applications and software complexity. To the best of our knowledge, we are the first to evaluate the performance of a dynamic DFT framework in such depth. We also compare the performance of libdft with the results reported in selected related studies, and proceed to evaluate the effectiveness of the various optimizations and design decisions we took. We close this section with an evaluation of the performance and effectiveness of the libdft-powered DTA tool.

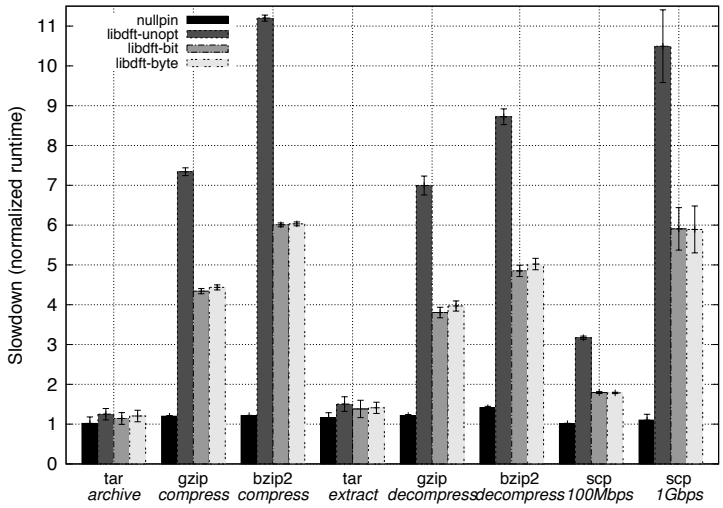


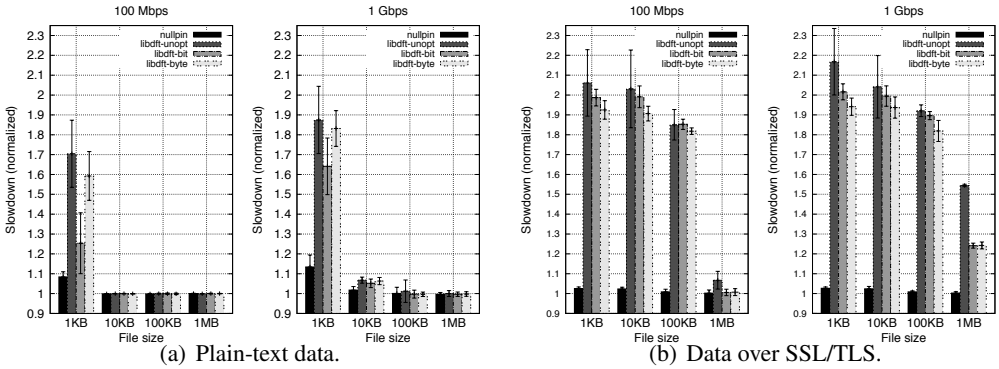
Fig. 3.4: The slowdown imposed by Pin and libdft when running four common Unix command-line utilities.

Our testbed for the evaluation consisted of two identical hosts, equipped with two 2.66 GHz quad core Intel Xeon X5500 CPUs and 24GB of RAM each, running Linux (Debian "squeeze" with kernel version 2.6.32). The version of Pin used during the evaluation was 2.9 (build 39586). While conducting our experiments all hosts were idle.

3.4.1 Performance

We developed four simple Pintools to aid us in evaluating libdft. The first is *nullpin*, which essentially runs a process using Pin without any form of instrumentation or analysis. This tool measures the overhead imposed by Pin's runtime environment alone. The second and third, namely *libdft-*{*bit, byte*}, utilize libdft for applying DFT on the application being executed, and use bit-sized and byte-sized tags respectively. These tools measure the overhead of libdft when employing single assignment and branch-less propagation. Finally, in order to demonstrate the efficacy of our design choices, we also evaluated *libdft-unopt*, which approximates the behavior of an unoptimized DFT framework. Specifically, it does not employ any optimization scheme and its analysis routines are not inlined, effectively resembling the impact of lax tag propagation.[1] Note that since the performance of libdft does not depend on the existence or amount of tagged data, none of our tools uses

[1]The body of the analysis routines used in libdft-unopt remains highly condensed, since they are the same routines that we use in libdft-{bit, byte}. Hence, the overhead measured with this tool gives a lower bound of an unoptimized DFT implementation.



(a) Plain-text data. (b) Data over SSL/TLS.

Fig. 3.5: The slowdown incurred by Pin and libdft on the Apache web server when serving static HTML files of different sizes.

any of the API functions for customizing the applied DFT.

Utilities The goal of our first benchmark was to quantify the performance of libdft with commonly used Unix utilities. For this experiment, we used the GNU versions of `tar`, `gzip`, and `bzip2`, as well as `scp` from the OpenSSH package. We selected these applications because they represent different workloads. `tar` performs mostly I/O, while `gzip` and `bzip2` are CPU-bound applications. In between, `scp` is both I/O driven and CPU intensive.

We run all the tools natively, and using our four tools, and measured their execution time with Unix `time` utility.

libdft-bit imposes a slowdown that ranges between 1.14x and 6x (average 3.65x), while libdft-byte ranges between 1.20x and 6.03x (average 3.72x). Pin alone imposes an 1.17x slowdown (this is Pin's baseline). Overall, the more CPU-bound an application is, the larger the impact of libdft. For instance, `bzip2` is the most CPU intensive and `tar` the least, representing the worst and best performance of libdft. I/O operations have a positive effect on DFT. This is also confirmed by the overhead of `scp` over an 1Gbps link, which is higher when compared with the 100Mbps case.

We surmise that nullpin and libdft perform worse when the limiting factor is not the I/O, because the respective latency hides the translation and tag propagation overhead. Indeed, when utilizing the 1Gbps link, the bottleneck shifts to the CPU, greatly reducing `scp`'s throughput. Also note that byte-sized tags impose an additional overhead of 1.9%, which stems from the management cost of the tagmap segments and memory shadowing (see Section 3.3.1.2). Compared with an unoptimized implementation, libdft with all our optimizations performs 3.64% to 46.37% faster.

Apache The second set of experiments calculates the performance slowdown of libdft when applied on larger and more complex software. Specifically, we investigate how libdft behaves when instrumenting the commonly used Apache web server. We used Apache v2.2.16 and configured it to pre-fork all the worker processes (pre-forking is a standard multiprocessing Apache module), in order to avoid high fluctuations in performance, due to Apache forking extra processes for handling the incoming requests at the beginning of our experiments. All other options were left to their default setting. We measured Apache's performance using its own utility `ab` and static HTML files of different size. In particular, we chose files with sizes of 1KB, 10KB, 100KB, and 1MB, and once again run the server natively and with our four tools. We also tested Apache over different network links, first over an 100Mbps link and then over an 1Gbps link, as well as with and without SSL/TLS encryption.

Figure 3.5(a) illustrates the results for running Apache, and transferring data in plaintext. We observe that as the size of the file being served increases, libdft's overhead diminishes. Similarly to our previous experiment, the time Apache spends performing I/O hides our overhead. As a result, libdft has negligible performance impact when Apache is serving files larger than 10KB at 100Mbps and 100KB at 1Gbps. In antithesis, libdft imposes an 1.25x/1.64x slowdown with 1KB files at 100Mbps/1Gbps when using bit-sized tags. The overhead of byte-sized tagging becomes more evident with smaller files because more requests are served by Apache, also increasing the number of `mmap` calls performed. This leads to higher tagmap management overhead, as segments need to be frequently allocated and freed. We anticipate that we can amortize this extra overhead by releasing segments more lazily, as we may have to re-allocate them soon after.

Figure 3.5(b) shows the results of conducting the same experiments, but this time using SSL/TLS encryption. We notice that libdft has larger impact when running on top of SSL. In the 100Mbps scenario, the slowdown becomes negligible only for files larger than 1MB, whereas at 1Gbps libdft imposes an 1.24x slowdown even with 1MB files. The reason behind this behavior is that the intensive cryptographic operations performed by SSL make the server CPU-bound.

Interestingly, libdft-byte performs 3% better (on average) than libdft-bit. In order to better understand this behavior, we analyzed the mix of the executed instructions and observed that serving a web page over SSL results in an increased number of XFER-type instrumentations, where one of

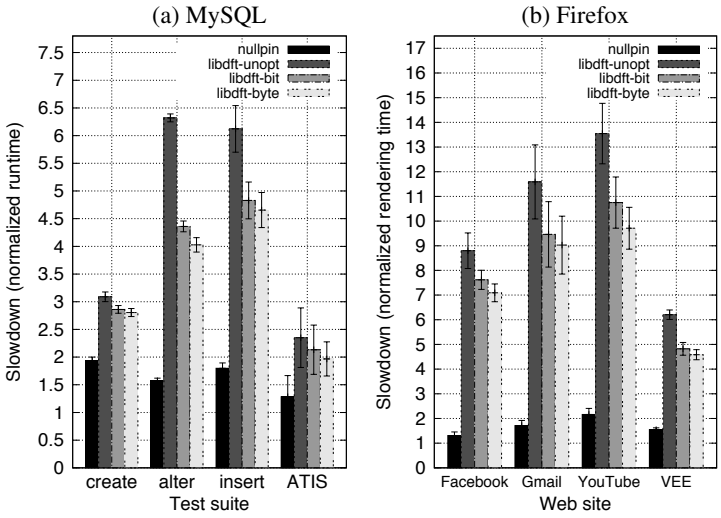


Fig. 3.6: The overhead of Pin and libdft when running MySQL and Firefox.

the two operands is memory.[2] *Tag propagation code that corresponds to data transfers is more expensive in the case of bit-sized tags than the case of byte-sized tags.* This is because the body of the respective *r2m*, *m2r*, and *m2m* analysis routines contains more instructions, due to the elaborate bit operations that are necessary for asserting only specific bits in the tagmap (see Figure 3.3(a) and 3.3(b) in Section 3.3.2.2).

MySQL In Figure 3.6(a), we present the results from evaluating MySQL DB server. We used MySQL v5.1.49 and its own benchmark suite (`sql-bench`). The suite consists of four different tests, which assess the completion time of various DB operations like table creation and modification, data selection and insertion, etc We notice that the average slowdown incurred by Pin's instrumentation alone is 1.64x, which is higher than the overhead observed when running smaller utilities (see Figure 3.4). The increased overhead is attributed to the significantly larger size of MySQL's codebase, which applies pressure on Pin's JIT compiler and code cache.

As far as libdft is concerned, the average slowdown on the five test suites was 3.36x when using byte-sized tags, and 3.55x when using bit-sized tags. Similarly to our previous experiments, libdft's overhead became more pronounced with more complex and CPU intensive tasks.

Firefox After evaluating two of the most popular servers, we tested libdft with the Firefox web browser that has even larger and more complex codebase, and complements our evaluation of client-

[2]SSL makes heavy use of instructions like `MOVS`, `STOS`, `MOVSX`, and `MOVZX`.

Application	Vulnerability	SecurityFocus ID
wu-ftpd v2.6.0	Format string	BID 1387
ATPhttpd v0.4	Stack overflow	BID 5215
WsMp3 v0.0.2	Heap corruption	BID 6240
ProFTPD v1.3.3	Stack overflow	BID 44562

Table 3.1: libdft-DTA successfully prevented the listed attacks.

side software that started with the smaller utilities. We used Mozilla Firefox v3.6.18 to access the three most popular web sites according to Alexa's Top 500 (`http://www.alexa.com/topsites`). Figure 3.6(b) illustrates the results for this experiment. We see that the average slowdown under nullpin, libdft-bit, and libdft-byte, was 1.68x, 8.16x, and 7.06x respectively. The overhead is relatively low when accessing mostly static content web pages (VEE), while it increases significantly when accessing media-rich sites (YouTube), and sites that depend heavily on JavaScript (JS) like Gmail. Note that for Facebook and Gmail, we accessed and measured a real profile page and not the log in screen. The effect of optimizations ranges between 18.64% and 29.68%.

Next, we benchmarked the JS engine of Firefox using the Dromaeo test suite (`http://dromaeo.com/?recommended`). Dromaeo measures the speed of specific JS operations, so there is no I/O or network lag, as in the page loading benchmark. We observed that the slowdown incurred by both nullpin and libdft was considerably higher, with an average of 3.2x for nullpin, and 14.52x/13.9x for libdft-bit/libdft-byte respectively. While increased overhead was expected because this benchmark is also CPU-bound, the slowdown is significantly larger from previous CPU intensive experiments. *It seems that since JS is interpreted and subsequently "jitted" by the browser's runtime, it interferes with the translation and optimizations performed by Pin's JIT engine, thus leading to excessive runtime overheads.* In fact, this implies that significant overheads would emerge whenever DBI is combined with an interpreting runtime environment such as JS. Regardless, when using the browser to access common web sites, we do not suffer the performance decline observed in the Dromaeo benchmark.

3.4.2 The libdft-DTA Tool

The purpose of developing the libdft-DTA tool was not to provide a solid DTA solution, but to demonstrate that our API can be used to easily and quickly develop otherwise complex tools. Nevertheless, we tested its effectiveness using the set of exploits listed in Table 3.1. In all cases, it successfully detected and prevented the exploitation of the corresponding application.

We also evaluated the performance of the tool using `scp`, Apache, and MySQL, and compare it with libdft's baseline performance. We observe that the additional overhead imposed by the tool is negligible ($\leq$ +7%) over the baseline overhead of DFT. While we cannot claim that all libdft-based tools will have such low additional overhead, our results demonstrate that libdft can be customized to implement problem-specific instances of DFT efficiently.

Part III

DFT Optimizations

From this chapter, we introduce two important optimization schemes that enhances the performance of libdft our baseline DFT framework. Our first approach TFA is the approach that segregates the taint tracking logic from the original program logic to restore higher level DFT semantics and apply techniques inspired by compiler optimizations. The second approach executes program and taint tracking logics in parallel to further reduce runtime overhead. This approach mitigates the issue of high communication overhead between two threads (or processes) running from different CPU cores. Although these approaches are currently implemented for libdft, the core concepts are general and can easily ported to other DFT frameworks for similar amount of performance improvement.

Chapter 4

Compiler Optimizations for DFT

4.1 DFT Optimizations with TFA

Taint Flow Algebra (TFA) is a novel optimization approach to DFT, based on combining static and dynamic analysis, which significantly improves its performance. Our methodology is based on separating program logic from taint tracking logic, extracting the semantics of the latter, and representing them using our own intermediate representation of TFA. We apply multiple code optimization techniques to eliminate redundant tracking logic and minimize interference with the target program, in a manner similar to an optimizing compiler. We draw on the rich theory on basic block optimization and data flow analysis, done in the context of compilers, to argue the safety and correctness of our algorithm using a formal framework.

We evaluate the correctness and performance of our methodology by employing libdft and show that the code generated by our analysis behaves correctly when performing dynamic taint analysis (DTA) [Newsome and Song, 2005]. Our results indicate performance gains as high as $2.23\times$, and an average of $1.72\times$ across all tested applications over libdft.

The main contributions of Taint Flow Algebra (TFA) are

- We demonstrate a methodology for segregating program logic from data tracking logic, and optimizing the latter separately using a combination of static and dynamic analysis techniques. Our approach is generic and can benefit most binary-only DFT approaches.

- We define a DFT specific intermeidate representation that we use to represent the tracking

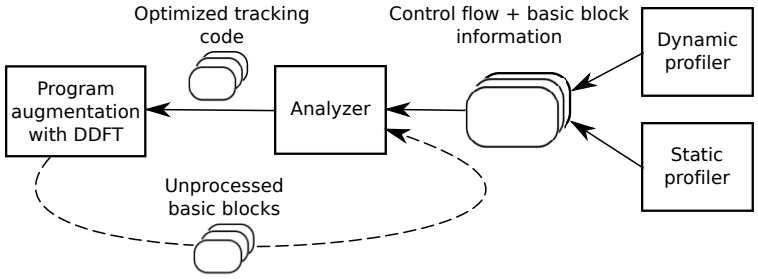


Fig. 4.1: TFA Approach Overview.

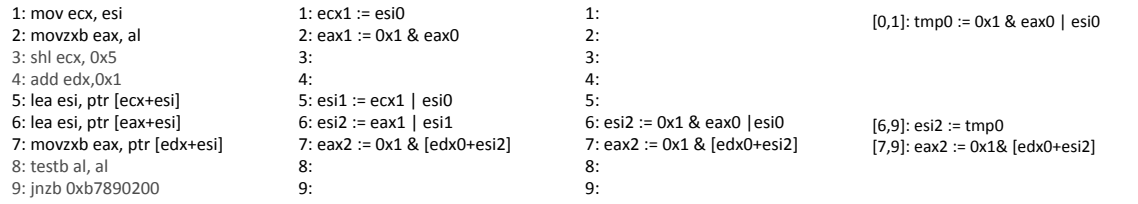
logic of programs to assist the analysis process.

- We demonstrate the application of classical compiler optimization techniques to the defined TFA. For instance, dead code elimination, copy propagation, and algebraic simplification are implicitly performed in the course of our static analysis.

- The resulting optimized taint tracking code is up to $2.23\times$ faster than unoptimized data tracking (libdft). Furthermore, our optimizations are orthogonal to other performance improvement approaches, and can provide additive benefits.

4.2 Taint Flow Algebra (TFA) Overview

Figure 4.1 shows a high-level overview of our approach for optimizing data tracking. We start by dynamically and statically profiling the target application to extract its basic blocks and control flow information. A basic block of code consists of a sequence of instructions that has only one entry point and, in our case, a single exit point. This means that no instruction within a basic block is the target of a jump or branch instruction, and the block is only exited after its last instruction executes. These properties are desirable for various types of analysis, like the ones performed by compilers. The control flow information describes how the basic blocks are linked. It is frequently impossible to obtain the complete control flow graph (CFG) for an entire program, but fortunately our analysis does not require a complete CFG. Nonetheless, the combination of dynamic and static profiling provides us with a significant part of the CFG, including the part that dominates in terms of execution time, and would benefit the most from optimization.



(a) Original x86 instructions (b) TFA representation (c) TFA optimization (d) Live range realignment

Fig. 4.2: A block of x86 instructions as it is transformed by our analysis.

The analyzer receives the profiler information and extracts data dependencies from the code, separating program from data tracking logic. It then transforms the latter to an internal representation, based on the Taint Flow Algebra (TFA) described in Section 4.3.1, which is highly amenable to various optimizations. The optimizations performed by the analyzer are described in Section 4.3, and include classic compiler optimizations like *dead-code elimination* and *copy propagation*. Our goal is to remove redundant tracking operations, and reduce the number of locations where tracking code is inserted. Finally, the analyzer emits optimized tracking code, which is applied on the application. Note that the type of tracking code generated depends on the original tracking implementation to be optimized. In our implementation, as it operates on binary programs the analyzer produces primitive C code, which can be compiled and inserted into the application using a DBI framework such as Pin [Luk *et al.*, 2005].

It is possible that the profiling of the program is incomplete, so when running it through a DBI framework we may encounter basic blocks that have not been yet optimized/analyzed, which will be instrumented with unoptimized data tracking code. We can assist the profiler by exporting the unprocessed blocks, so that they be added to the analysis. This process establishes a feedback loop between the runtime and the analyzer, as shown in Figure 4.1.

4.3 Taint Flow Algebra (TFA) Static Analysis

This section begins a formal definition of the optimizations performed. Throughout this section, we consider the basic block (BBL), a set of instructions with a single entry and exit, as the primitive unit

reg : *variable-name unsigned*
mem : *'[' mem-expr ']'*
var : *reg | mem | const*

binary-opr : *'|' | '&'*
unary-opr : *'~'*
assign-opr : *':='*
rng-map : *'r (' var | expr | statement ')'*

mem-expr : *(reg |const) |{ '×' | '+' | '−' (reg|cost)};*
expr : *var | expr {var binary-opr} | '(' expr ')'*
| unary-opr expr
statement : *(reg | mem) assign-opr expr*

Fig. 4.3: The abstract syntax tree used by our Taint Flow Algebra.

for profiling, analyzing, and executing a binary. [1] To assist the reader better comprehend the various steps of this process, as we present them in detail below, we employ the code sample depicted in Figure 4.2. It demonstrates how we process a given *x86* block of binary code as it goes through the different analyses. As a result of our optimization, the five taint propagation operations required originally (Figure 4.2(a)) are reduced to three, inserted in two locations (Figure 4.2(d)).

4.3.1 Definition of a Taint Flow Algebra

TFA offers a machine independent representation of the data flow tracking logic that exposes optimization opportunities. It expresses a system with memory and an infinite number of registers, so it can accommodate a variety of instruction-set architectures (ISAs). Figure 4.3 shows its abstract syntax tree (AST) in BNF form.

It supports three different types of variables: constants, registers, and memory. The latter two correspond to the tags that hold the taint markings for the underlying architecture, while constants are primarily used to assert or clear tags. For example, when using a single-bit mark for each byte of data, the constant `0xf` (`1111` in binary) represents four tainted bytes. Other tag sizes are also supported by simply modifying the constants and variable size. Register variables are versioned (i.e., a new register version is generated on update) to help us identify live ranges and redundant

[1] To be persistent across separate analysis stages, a basic block identifier (BBID) is computed by hashing the full path name of the binary image it belongs to and adding its offset from the image's base address. This facilitates TFA to identify blocks that begin from different entry points but exit at the same point as separate ones, and makes each block have its own optimization result.

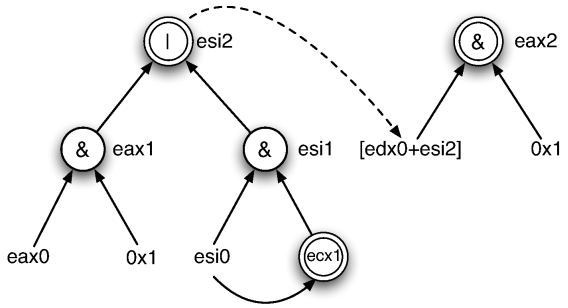


Fig. 4.4: The taint-map for Figure 4.2(b), viewed as a direct acyclic graph (DAG).

operations, while memory follows a flat model with array-like index addressing. Also, all variables in a single statement are of the same size, and we use the constants combined with the AND (bitwise $\&$) and NOR (unary $\sim$) operators to extend or downsize a variable due to casting. We express data dependencies among variables using the OR operator (bitwise $|$), and determine operator precedence using parentheses ("(", ")").

Unlike other general representations like LLVM-IR or gcc's GIMPLE, our TFA does not require primitives to express control transfers made by branch instructions or function calls. Instead, we employ a separate optimization stage to handle inter-block dependencies. As a result, branch choices solely depend on the original execution of the program. A *range operator* (`rng-map`) also differentiates the TFA from other representations, allowing us to specify live ranges for its various elements (i.e., statements, expressions, and variables), so we can move certain statements in different positions and still correctly preserve code semantics. Constant variables are valid for entire blocks, while register variables are live from their previous till their next definition. In the case of memory variables, we also need to consider the registers and constants used to calculate their address. Hence, we determine their liveness range by looking for ranges, where both the variables used in addressing and the memory variable itself are valid concurrently. The liveness range for expressions and statements can be obtained by combining the ranges of their contained elements. Statements in Figure 4.2(d) are prefixed with a liveness range gained by applying the range operator($r(\cdot)$) to the statements in Figure 4.2(c).

4.3.2 Data Dependencies Extraction and TFA Representation

Our analysis begins by parsing a basic block of code and representing it in our TFA. This stage is specific to the type of code. For example, the x86 binary code in Figure 4.2(a) is represented as shown in Figure 4.2(b). To "translate" an instruction to TFA, we first determine the data dependencies it defines. In other words, we extract the data tracking logic or taint tracking semantics from the block's instructions. For instance, the `MOV` instruction propagates taint from source to destination, and the *ALU* instruction family (`ADD`, `SUB`, `XOR`, `DIV`, etc) tags the destination operand, if one of source operands is also tagged. Even though it may seem like a straightforward process, *in practice different taint-tracking tools may adopt slightly divergent semantics*. For instance, sign extending a tainted 16-bit number to 32 bits may taint the entire result [Kemerlis *et al.*, 2012], or just the lower 16 bits [Portokalidis *et al.*, 2006]. Currently, we have adopted the interpretation used by libdft (also used for our prototype implementation), but it can be effortlessly modified to cater to other implementations or problem domains.

While extracting dependencies, we can already discard some instructions, as most DFT systems also do. For instance, arithmetic operations with constants (lines 2 and 3 in Figure 4.2(a)) do not propagate taint. We also handle language idioms like `xor eax,eax` and `sub eax,eax`, which are used to clear registers, by clearing the corresponding tags (e.g., `t(eax)`$\leftarrow$`0`). At this stage, we also cast operands of different sizes to the same width using masking and unmasking operations (i.e., using the logical operators and constants). For instance, instruction 7 in Figure 4.2(a) transfers one byte from `[edx+esi]` to `eax`. We use the constant `0x1` to mask the taint propagation, ensuring that the higher bytes of the destination are correctly marked as clean.

The results of data dependency analysis (Figure 4.2(b)) are stored in a hash-table data structure, which we call the *taint-map*. The taint-map holds one entry for every TFA statement, using the destination operand as *key* and the *right hand-side* (*rhs*) expression as *value*. Combining all its entries eventually presents us with a directed acyclic graph (DAG), where inputs are the children of outputs as shown in Figure 4.4. The solid lines in the figure represent data dependencies within a statement, while dashed lines represent dependencies between variables.The leaf nodes of the DAG are input variables, and double-lined nodes are output variables. Finally, during this phase, we can introduce different dependency interpretations based on tracking policies such as implicit data flow tracking, and schemes like pointer tainting [Yin *et al.*, 2007] or heap pointer tracking [Venkataramani *et al.*,

2008].

Inner Optimization After obtaining the TFA representation of a block, we process it to identify the operands used as inputs and outputs. The first version of every operand on the *rhs* of an assignment is marked as input. Similarly, the greatest version of an operand on the *left-hand side* (*lhs*) of an assignment is marked as output. From Figure 4.2(b), {`esi0`, `eax0`, `[edx0+esi2]`} are inputs, whereas {`ecx1`, `eax2`, `esi2`} are outputs. This process enables us to perform rudimentary dead code elimination [Aho *et al.*, 2006], to discard taint propagation operations that do not affect output variables from the basic block. From our toy example, the taint operation in line 5 of Figure 4.2(b) can be removed as the taintedness of `esi1` solely depends on prior version of the same register variable (i.e.,`esi0`.)

4.3.3 Incorporating Control Flow Information - Outer Optimization

We make use of the control flow information collected during the code profiling stage to extend *the inputs/outputs identification* to span multiple chained basic blocks. For example, if the CFG points out that BBL `Block0` is directly followed by either `Block1` or `Block2`, and no other BBLs, we can use the inputs/outputs lists of those BBLs to eliminate some of `Block0`'s output variables, if they are overwritten without being used as inputs in either `Block1` or `Block2`. This implements a type of (static) dataflow analysis (or live variable analysis) common in compiler optimizations, and allows us to purge more redundant tracking operations, based on the new, smaller outputs lists. *Our analysis differs from commonly used algorithms in that the available CFG used may be incomplete, and the BBLs we use, can span across the boundary of functions and even libraries.*

In a case where we cannot identify all successor blocks due to indirect calls or jumps, we handle it conservatively by considering all output variables of a BBL as live ones. This modification should not have much impact on performance, since a profiling result for SPEC CPU2000 benchmark suite shows that about 95.7% of BBLs are ended with deterministic control transfers such as direct and conditional jumps which only allow one and two successor nodes respectively.

Theorem 1 formally addresses the soundness of outer analysis whose correctness can be proved using a semi-lattice framework [Aho *et al.*, 2006], modified to cover cases where the CFG is incomplete. Due to space constraints, we accommodate the proof in Appendix 1.2.1.

Theorem 1. ***Soundness of outer analysis:*** *Outer analysis (live variable analysis) with incomplete CFG converges and is safe.*

In our toy example (Figure 4.2(b)), the output variable `ecx1` taints `esi1` in line 5, but our analysis determines that it is not used in any of subsequent blocks. Therefore, we remove `ecx1` and the corresponding propagation operation. Note that taint propagations in line 5 and line 6 preserve original taint semantics without `ecx1` as their taintedness solely depends on `esi0`.

4.3.4 Pruning Redundant Expressions and Merging Statements

Copy propagation is the process of replacing the occurrences of targets of direct assignments (e.g., `eax1` in Figure 4.2(b)) with the expression assigned to them (e.g., `0x1` & `eax0`), where they appear in other statements (e.g., line 6 in Figure 4.2(b)). We perform this process by recursively substituting variables in the *rhs* expressions of assignments, until they solely contain variables from the inputs and outputs lists. We use the taint-map to quickly look for targets of assignments. The result is an updated taint-map that contains entries that use a variable from BBL outputs as key, while its value is the taint expression assigned to the key. This can also be seen as a process of directly connecting output and input nodes from DAG representation of a block (i.e., Figure 4.4) by eliminating intermediate nodes. The number of tracking operations needed is greatly reduced, as shown in Figure 4.2(c). Even though, some of the generated assignments are more complex, they will not generate longer expressions than the original while they also reduce the number of locations where propagation code needs to be inserted. Due to its recursive nature, the running time of our substitution algorithm increases exponentially with the size of BBL. We alleviate this problem by memoizing previously visited variables.

Along the way, we also perform *algebraic simplification* on arithmetic operations such as (`0xffff` & (`0x00ff` & `eax0`) ⇒ (`0x00ff` & `eax0`). This optimization may be also performed in later stages from the deployment platform, like the DBI framework. However, we make no assumptions about the underlying framework, so we proactively exploit optimization opportunities like this, whenever they are feasible.

Resolving Range Violations in Merged Assignments Substituting variables can raise issues, when variables in a statement cannot be live (i.e., valid) concurrently. For example, `esi2` and `eax0`

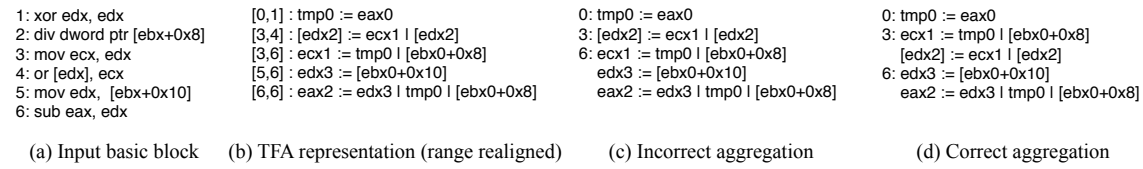
in instruction six of Figure 4.2(c) have ranges of [5,9] and [0,1] respectively, which we calculate and express using the range operator (`r(·)`), and are not valid concurrently. When a statement cannot yield a common range for their contained elements, we introduce temporary variables to resolve the conflict. For example, in Figure 4.2(d) we use `tmp0` to resolve the range conflict. Note that the range correction should be made in a way that the number of instrumentation locations is kept to the least possible, and should not result in more tracking operations than the original representation. Theorem 2 directly states this, its proof is provided in Appendix 1.2.2.

Theorem 2. ***Efficiency of the TFA optimization****: The TFA optimization always produces less, or an equal number of, tracking statements than the original representation, for any basic block.*

One can argue that showing to have less statements than the original may not be sufficient, since longer statements containing many variables could, in theory, be translated into more machine instructions. Even though precisely estimating the outcome of our optimization in the instruction level is not an easy task, we can still provide a *corollary* to the theorem that states that the optimization's results will always have less or the same number of variables than the original representation as a whole. This can be proven in similar way to Theorem 2. However, the theorem does not cover some corner-cases, where the TFA representation has more statements than the original instrumentation. For instance, in the presence of ISA specific instructions, such as x86's `xchg` or `cmpxchg` instructions which requires two or more statements to be expressed in TFA whereas a baseline system can have a specialized and rather efficient interpretation. We detect such cases and retain the original propagation logic.

4.3.4.1 Minimizing Instrumentation Interference with Aggregation

By default the TFA optimization produces *scattered* statements. This leaves the merged statements near the location where their target is actually being updated. As the range operator can yield multiple valid locations for each statement, we can adopt different policies to group statements, in order to minimize the instrumentation overhead. *Aggregation* aims to group statements into larger instrumentation units, so as to further reduce the number of locations where code is injected into the original program. Aggregation can be particularly effective when used with DFT frameworks



(a) Input basic block (b) TFA representation (range realigned) (c) Incorrect aggregation (d) Correct aggregation

Fig. 4.5: Aggregation example with range violations.

where the fixed overhead associated with every instrumentation is high.[2] In the example shown in Figure 4.2(d), aggregation will combine the second and third statements into a single block with an aggregated liveness range of [7,9].

Range Violations among Statements In section 4.3.4, we saw how we address invalid expressions produced because the elements of an expression are valid in different code ranges. Less obvious range violation errors may also occur with *aggregation*. For example, consider the BB in Figure 4.5(a), which after our analysis produces the statements in Figure 4.5(b) (prefixed with their calculated live ranges). Both Figures 4.5(c) and 4.5(d) are valid groupings consistent to each statement's liveness range, but the first will incorrectly propagate taint. That is because in Figure 4.5(c), the second statement uses `ecx1` before it is defined in the third statement. This *define/use violation* occurs because the aggregation algorithm greedily attempts to maximize the size of instrumentation unit. Also, ordering of the second and third statement causes a *memory alias violation*. In the original order shown in Figure 4.5(c), the memory variable `[ebx0+0x8]` is used before memory variable `[edx2]` is defined. Since we cannot determine if these two variables point to the same location or not, all memory variables are considered to be dependent on each other.

To address these issues, we impose dependencies between statements using a variable and the statement defining it. We implement these dependencies by topologically sorting the taint-map data structure, and relocating instructions based on their order. As it is shown by the corrected aggregation in Figure 4.5(d), we increase neither the number of instrumentation units nor the code size as a whole, but only redistribute the statements.

[2]For most DBI/VM based frameworks each instrumentation requires additional instructions for context switching. For instance, Pin [**?**] introduces three additional instructions per instrumentation location for analysis routines that can be inlined and 12 otherwise.

4.3.5 Code Generation

To apply the results of our analysis on a DFT framework, we need to transform the statements from TFA to a language that the framework understands. We do so using both the statements produced pre- and post-aggregation, namely *TFA scatter* and *TFA aggregation*. The reason for doing so is that the number of instrumentation locations and the size of the routines that need to be injected affect varying frameworks in different ways (i.e., some may prefer one instead of the other). Consequently, to implement the code generation back-end, we need to know certain features of the deployment platform, like *(i)* architectural properties such as the ISA type (e.g., *x86* or *x86_64*), the underlying OS, and the instrumentation framework employed (e.g., PIN, DynamioRIO, QEMU, etc), and *ii)* DFT characteristics such as tag size, tracking granularity, and any particular rules used for determining data dependencies. Most of all, the produced code needs to use the DFT framework's primitives for accessing and updating tags for different variable types. For our prototype implementation, we utilized the libdft.

Chapter 5

Parallel Execution of DFT

5.1 Parallel (Decoupled) Execution of DFT with ShadowReplica

ShadowReplica offloads the DFT analysis code alone to another execution thread. It instruments the application to collect all the information required to run the analysis independently, and communicate the information to a thread running the analysis logic alone. ShadowReplica's main contribution is an off-line application analysis phase that utilizes both static and dynamic analysis approaches to generate optimized code for collecting information from the application, greatly reducing the amount of data that we need to communicate. For running DFT independently from the application, such data include dynamically computed information like memory addresses used by the program, control flow decisions, and certain operating system (OS) events like system calls and signals. We focus on the first two that consist the bulk of information. For addresses, we exploit memory locality to only communicate a smaller set of them, and have the DFT code reconstruct the rest based on information extracted by the off-line analysis. For control flow decisions, we exploit the fact that most branches have a binary outcome and introduce an intelligent encoding of the information sent to DFT to skip the most frequent ones.

DFT is run in parallel by a second shadow thread that is spawned for each application thread, and the two communicate using a shared data structure. The design of this structure is crucial to avoid the poor cache performance issues suffered by previous work. We adopt a lock-free ring buffer structure, consisting of multiple buffers (N-way buffering scheme [Zhao *et al.*, 2008]). After experimentation, we identified the optimal size for the sub-buffers of the structure, so that when two

threads are scheduled on different cores on the same CPU die, we achieve a high cache-hit rate for the shared L3 cache and a low-eviction rate on each core's L1 and L2 caches. The latter is caused when two cores are concurrently read/write in memory that occupies the same cache line in their L1/L2 caches.

The code implementing DFT is generated during off-line analysis as a C program, and includes a series of compiler-inspired optimizations that accelerate DFT by ignoring dependencies that have no effect or cancel out each other (i.e., TFA). Besides the tag propagation logic, this code also includes per-basic block functionality to receive all data required (e.g., dynamic addresses and branch decisions). Note that even though the code is in C, it is generated based on the analysis of the binary without the need for application source code. The implementation is also generic, meaning that it can accommodate different tag sizes in shadow memory. Such flexibility is partly allowed by decoupling DFT from execution. We implemented ShadowReplica using Intel's Pin dynamic binary instrumentation (DBI) framework [Luk *et al.*, 2005] to instrument the application and collect the data required to decouple DFT. Shadow threads running the DFT code run natively (without Pin), but in the same address space. Similar to libdft and TFA, it also implements dynamic taint analysis [Newsome and Song, 2005] protection from memory corruption exploits. Our evaluation shows that compared with an already optimized in-lined DFT implementations (libdft and TFA), it is extremely effective in accelerating *both* the application and DFT, but also using less CPU cycles. In other words, we do not sacrifice the spare cores to accelerate DFT, but exploit parallelization to improve the efficiency of DFT in all fronts. ShadowReplica is on average $\sim$2.3$\times$ faster than in-lined DTA (TFA) when running the SPEC2006 benchmark ($\sim$2.75$\times$ slowdown over native execution). We observed similar speed ups with command-line utilities, like `bzip2` and `tar`, and the Apache and MySQL servers. We also discovered that with ShadowReplica applying DFT requires less CPU cycles than the in-lined case, reaching a 30% reduction in the 401.bzip2 benchmark.

5.2 ShadowReplica Overview

Figure 5.1 depicts the architecture of ShadowReplica, which comprises of two stages. The first stage, shown in the left of the figure involves profiling an application both statically and dynamically to extract code blocks, or BBLs, and control-flow information (CFG+). The latter includes a partial

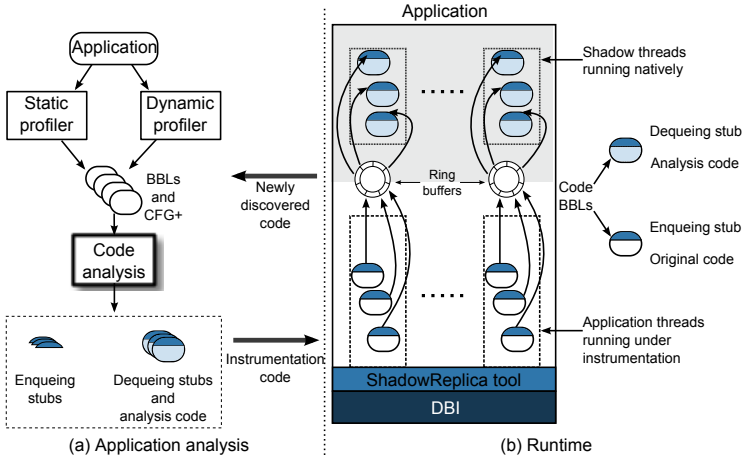


Fig. 5.1: The architecture of ShadowReplica

control-flow graph showing how the extracted BBLs are connected, and frequency data indicating which branches are taken more frequently than others.

This data is processed to generate optimized code to be injected in the application, and code for running the analysis in parallel. The first contains code stubs that enqueue the information required to decouple DFT in a shared data structure. Note that ShadowReplica does not naively generate code for enqueueing everything, but ensures that only information that has potentially changed since the previous executed block are enqueued. This is one of our main contributions, and problems for previous works [Nightingale *et al.*, 2008; Wallace and Hazelwood, 2007; Zhao *et al.*, 2008]. The second includes code stubs that dequeue information along with the analysis code.

The generated code is passed to the runtime component of ShadowReplica, shown in Figiure 5.1 (b). We again utilize a Pin DBI framework that allows us to inject the enqueueing stubs in the application in an efficient manner and with flexibility (i.e., on arbitrary instructions of a binary). Our motivation for using a DBI is that it allows us to apply ShadowReplica on unmodified binary applications, and it enables different analyses, security related or others, by offering the ability to *interfere* with the application at the lowest possible level.

Application threads are executing over the DBI and our tool, which inject the enqueueing stubs. We will refer to an application thread as the primary. For each primary, we spawn a shadow thread that will run the analysis code, which we will refer to as the secondary. While both threads are in the same address space, applications threads are running over the Pin DBIs VMM, but shadow threads

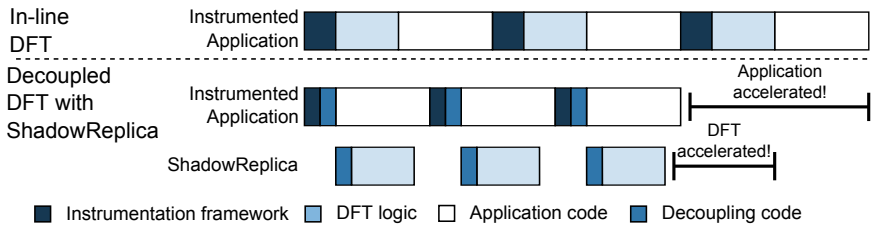


Fig. 5.2: Inline vs.decoupled application of DFT application with ShadowReplica and binary instrumentation.

are executing natively, since the code generated in the first phase includes everything required to run the analysis.

Communication between primary and secondary threads is done through a ring-buffer structure optimized for multi-core architectures. The ring buffer is also used for the primary thread to synchronize with the secondary, when it is required that the analysis is complete before proceeding with execution. For instance, ensuring that integrity has not been compromised before allowing a certain system call or performing a computed branch.

Finally, we export any new BBLs and CFG edges that are discovered at runtime, which can be passed back for code analysis. Extending the coverage of our analysis means that we can generate optimal code for a larger part of the application. Note that our analysis also generates generic code for handling application code not discovered during profiling. This default code performs all necessary functionality, albeit slower than the optimized code generated for known BBLs and control-flow edges.

5.2.1 Inlined vs. Decoupled DFT

Dynamically applying DFT on binaries usually involves the use of a DBI framework or a virtual machine monitor (VMM) that will transparently extend the program being analyzed. Such frameworks enable us to inject code implementing DFT in binaries, by interleaving framework and DFT code with application code, as shown in Fig.5.2 (*in-line*) and libdft and TFA falls in this category.

ShadowReplica proposes an efficient approach for accelerating dynamic DFT and similar analyses by decoupling them from execution and utilizing spare CPU cores to run the instrumented application and DFT code in parallel. We replace the in-line DFT logic in the application with a

stub that *extracts* the minimal information required to independently perform the analysis in another thread, and *enqueues* the information in a shared data structure. The DFT code, which is running on a different CPU core, is prefixed with a consumer stub that *pulls out* the information and then *performs* the analysis.

Decoupling the analysis from execution enables us to run it completely independently and without involving the instrumentation framework, as illustrated in Fig. 5.2 (*decoupled*). Depending on the cost of the analysis (e.g., tracking implicit information flows is more costly than explicit flows), it can accelerate both application and analysis. In short, if I_i, A_i, and P_i are the instrumentation, analysis, and application code costs with in-line analysis, and I_d, A_d, P_d, E_d and D_d are the costs of instrumentation, analysis, application, enqueueing and dequeueing code (as defined in the above paragraph), then decoupling is efficient when:

$$I_i + A_i + P_i > max(I_d + P_d + E_d, A_d + D_d) \tag{5.1}$$

Essentially, decoupling is more efficient when the following two conditions are met: *(a)* if the cost of the in-line analysis is higher than the cost of extracting the information and enqueueing, and *(b)* if the cost of program execution combined with instrumentation interference is higher than dequeueing cost. Ha et al. [Ha *et al.*, 2009] provide a more extensive model of the costs and benefits involved with decoupling analysis.

Analyses that are bulky code-wise can experience even larger benefits because replacing them with more compact code, as decoupling does, exerts less pressure to the instrumentation framework, due to the smaller number of instructions that need to be interleaved with application code. For instance, when implementing DFT using binary instrumentation, the developer needs to take extra care to avoid large chunks of analysis code and conditional statements to achieve good performance (libdft). When decoupling DFT, we no longer have the same limitations, we could even use utility libraries and generally be more flexible.

We need to emphasize that ShadowReplica does *not* rely on complete execution replay [Chow *et al.*, 2008; Portokalidis *et al.*, 2010] or duplicating execution in other cores through speculative execution [Nightingale *et al.*, 2008; Wallace and Hazelwood, 2007]. So even though other cores may be utilized, it does not waste processing cycles. Application code runs exactly once, and the same stands for the analysis code that runs in parallel. The performance and energy conservation

benefits gained are solely due to exploiting the true parallelism offered by multi-cores, and being very efficient in collecting and communicating all the data required for the analysis to proceed independently.

5.3 ShadowReplica Static Analysis

We analyze the application off-line to generate optimized instrumentation code to be injected in primary threads, as well as a new program in C that implements the analysis in the secondary (based on the execution trace recorded by the primary). This section describes our methodology for doing so.

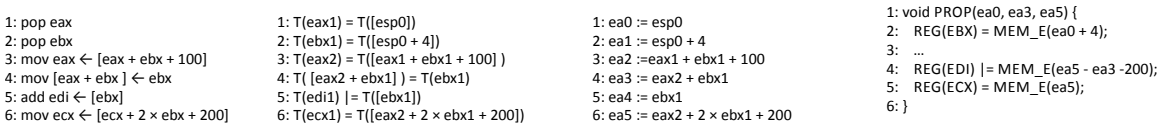
5.3.1 Application Profiling

The first step of application analysis involves profiling (Fig. 5.1(a)) to gather code and control-flow information. ShadowReplica uses both *static* and *dynamic* profiling. Currently, we perform static profiling using the IDA Pro [Hex-Rays, cited Aug 2013] disassembler, using its scripting API. To complement our data, we built a tool over the Pin [Luk *et al.*, 2005] DBI framework that dynamically collects information about a binary by executing it with various inputs (e.g., test inputs and benchmarks). New methodologies that improve code and CFG identification are orthogonal to our design and could be included alongside our toolkit with little effort.

Every BBL identified during profiling is assigned a unique id (BBID). BBIDs are calculated by combining the block's offset from the beginning of the executable image it belongs to with a hash of the image's text section to produce a 32-bit value. An example control-flow graph collected during profiling is shown in Fig. 5.4. During dynamic profiling, we also keep a counter of how many times each control-flow transfer is observed. In the example in Fig. 5.4, when executing BBL2, BBL4 was followed 9999 times while BBL3 only once.

5.3.2 Primary Code Generation

During code analysis we generate optimized code for the primary thread to enqueue information necessary for performing DFT in the secondary. The most frequently enqueued data are effective addresses used in memory accesses and control-flow transfers. This section discusses our approach



(a) x86 instruction (b) DFT representation (c) Distinct EAs (d) Propagation body

Fig. 5.3: Example of how a BBL is transformed during code analysis.

for minimizing the amount of data we need to send to the secondary.

5.3.2.1 Effective Address Recording

A naive approach would transfer all addresses involved in memory operations to the secondary, leading to excessive overhead. We developed a series of optimizations to reduce the number of addresses needed to be enqueued without compromising the soundness of the analysis. Fig. 5.3 will assist us in presenting our methods.

We begin by transforming the BBL in Fig. 5.3 (a) into the DFT-specific representation in Fig. 5.3 (b), which captures data dependencies and data tracking semantics, as defined in an intermediate representation of TFA.

Intra-block Optimization We search for effective addresses within a BBL that correlate with each other to identify the minimum set required that would correctly restore all of them in the secondary. For instance, we only need to enqueue one of `[esp0]` or `[esp0 + 4]` from Fig. 5.3 (b), as one can be derived from the other by adding/subtracting a constant offset. This search is greatly facilitated by the DFT transformation and register variables versioning, but it is applicable to all shadow memory analyses, as it only tries to eliminate related addresses.

DFT Optimization This optimization applies TFA approach to identify instructions, and consequently memory operands, which are not going to be used by the analysis in the secondary. For instance, in Fig. 5.3(b) we determine that the propagation in line 1 is redundant, as its destination operand (`eax1`) is overwritten later in line 3, before being referred by any other instruction. This allows us to ignore its memory operand `[esp0]`.

Inter-block Optimization We extend the scope of the intra-block optimization to cover multiple blocks connected by control transfers. This implements backward data-flow analysis [Aho *et al.*, 2006] with the partial CFG gathered during profiling. We begin by defining the input and output

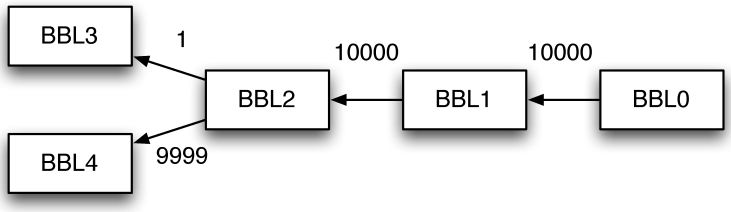


Fig. 5.4: Example CFG. Nodes represent basic blocks and edges are control transfers. During dynamic profiling, we count how many times each edge is followed (edge labels).

variables for each BBL. We then produce a list of input and output memory operands which are live before entering and when leaving a BBL. Using our representation, input memory operands are the ones with all of its constituent register variables in version 0, and output memory operands are the ones that have all of its constituent register variables at their maximum version. In our example in Fig. 5.3, the inputs list comprises of `[esp0]`, and the outputs list includes `[esp0]`, `[ebx1]`, `[eax2 + ebx1]`, and `[eax2 + 2 × ebx1 + 200]`. If all predecessors for the BBL contain `[esp0]` in their outputs list, the block can harmlessly exclude this from logging because the secondary still has a valid reference to the value. The optimization has greater effect as more inputs are found on the outputs lists of a block's predecessors.

Since we only have a partial CFG of the application, it is possible that at runtime we identify new execution paths that invalidate the backwards dataflow analysis. We tackle this issue by introducing two heuristics that make inter-block optimization more conservative. First, if we find any indirect jumps to a known BBL, we assume that others may also exist and exclude these blocks from optimization. Second, we assume that function entry point BBLs, may be reachable by other, unknown indirect calls, and we also exclude them. We consider these measures to be enough to cover most legitimate executions, but they are not formally complete, and as such may not cover applications that are buggy or malicious.

Having applied all, except the inter-block optimization, the number of effective addresses that need to be transferred for our example is reduced from six to three (from Fig. 5.3(c) to (d)). The effects of individual optimizations are discussed in Sec. 5.5.1.

5.3.2.2 Control Flow Recording

As in the previous section, a naive approach to ensure control flow replication would involve the primary enqueueing the BBID of every BBL being executed. However, simply doing this for all BBLs is *too* costly.

After examining how control-flow transitions are performed in x86 architectures, we identify three different types of instructions:
(a) direct jumps, (b) direct branches, and (c) indirect jumps. For direct jumps, BBIDs for successor BBLs can be excluded from logging, since there is only a single, fixed exit once execution enters into BBL0. For example, the transitions from BBL0 to BBL1, and then to BBL2 in Fig. 5.4 can be excluded. Direct branches can have two outcomes. They are either taken, or fall through where execution continues at the next instruction. We exploit this property to only enqueue a BBID, when the least frequent outcome, according to our dynamic profiling, occurs. For instance, when BBL3 follows BBL2 in Fig. 5.4. We use the *absence* of BBL3's id to signify that BBL4 followed as expected. Note that if a BBL has two predecessors and it is the most frequent path for only one of them, we log its BBID. Last, for indirect jumps we always record the BBID following them, since they are hard to predict. Fortunately, the number of such jumps are less compared to direct transfers.

Applying our approach on the example CFG from Fig. 5.4, we would only need to enqueue the id of BBL3 once. Obviously, this approach offers greater benefits when the profiling covers the segments of the application that are to run more frequently in practice. It does not require that it covers all code, but unknown code paths will not perform as well. We evaluate the effects of this optimization for various applications in Sec. 5.5.

5.3.2.3 Ring Buffer Fast Checking

For every enqueueing operation from the primary, we should check whether there is available space in the ring buffer to avoid an overflow. However, performing this check within the DBI is very costly, as it requires backing up the `eflags` register. We attempt to reduce the frequency of this operation by performing it selectively. For instance, every 100 BBLs. However, to correctly placing the checks in the presence of CFG loops, so that they are actually performed every 100 executed blocks, is challenging. We mitigated this problem by introducing an algorithm, which finds basic blocks that program execution is ensured to visit at most every k block executions. The problem

formally stated is as follows.

We are given a CFG defined as $C = (V, E)$. C is a weighted directed graph where V represents a set of basic blocks and E represents a set of edges that correspond to control transfers among blocks. We also have a weight function $w(v)$ that returns execution counts for $v \in V$. Given that we want to find a subset of vertices S such that:

- For a given parameter k, we can assure that the program execution will visit a node from S at most every k block executions.
- $\sum_{v \in S} w(v)$ is close to the minimum.

The above problem is identical to *the feedback vertex set problem* [Festa *et al.*, 1999], which is NP-complete when any non-cyclic paths from C is smaller than k. Thus, we can easily reduce our problem to this one and use one of its approximation approaches. Additionally, to take into consideration new execution paths not discovered during profiling, the secondary monitors the ring buffer and signals the primary, when it exceeds a safety threshold. Finally, we also allocate a write-protected memory page at the end of the available space in the ring buffer that will generate a page fault, which can be intercepted and handled, in the case that all other checks fails.

5.3.3 Secondary Code Generation

During the off-line analysis we generate C code that implements a program to dequeue information from the shared ring-buffer and implement DFT. Listing 5.1 contains a secondary code block generated for the example in Fig. 5.3.

5.3.3.1 Control Flow Restoration

Each code segment begins with a goto label (line 2) from Listing 5.1, which is used with the `goto` statement to transfer control to the segment. Control transfers are made by code appended to the segments. In this example, lines 21 $\sim$ 31 implement a direct branch. First, we check whether the ring buffer contains a valid effective address. The presence of an address instead of BBID (i.e., the *absence* of a BBID) indicates that the primary followed the more frequent path (BBID 0xef13a598), as we determined during code analysis (Sec. 5.3.2.2). Otherwise, the code most likely

did not take the branch and continued to the BBL we previously identified (in this case 0xef13a5ba). We do not blindly assume this, but rather check that this is indeed the case (line 25). We do this to accommodate unexpected control-flow transfers, such as the ones caused by signal delivery (Linux) or an exception (Windows). If an unexpected BBID is found, we perform a look up in a hash table that contains all BBLs identified during the analysis, and the result of the search becomes the target of the transfer in line 29. Note that unknown BBIDs point to a block handling the slow path. While a look up is costly, this path is visited rarely. We also use macros `likely()` and `unlikely()` (lines 5, 21, and 25) to hint the compiler to favor the likely part of the condition.

5.3.3.2 Optimized DFT

Each code segment generated performs tag propagation for the BBL it is associated with. Effective addresses are referenced directly within the ring buffer, and shadow memory is updated as required by code semantics. For each BBL, optimized tag propagation logic is generated based on TFA optimization.

The propagation code in lines $6 \sim 12$ is generated based on the example in Fig. 5.3. `rbuf()` is a macro that returns a value in the ring buffer relative to the current reading index. Ring buffer accesses correspond to `ea0`, `ea3` and `ea5` in Fig. 5.3(d). The `MEM_E()` macro in lines $8 \sim 11$ translates memory addresses from the real execution context to shadow memory locations, while `REG()` does the same for registers.

We exploit the fact that we can now run conditional statement fast, since we are running the analysis outside the DBI, to implement a simplified version of the FastPath (FP) optimization proposed by LIFT [Qin *et al.*, 2006]. This is done in line 8, where we first check whether any of the input and output variables involved in this block are tagged, before propagating empty tags.

Last, in line 15 we save the EA that the inter-block optimization determined it is also used by successor BBLs in the global arguments array `garg`, and progress the increasing index of the ring buffer in line 18.

```
1  /* BBL label */
2  BB_0xef13a586:
3
4  /* LIFT's FastPath(FP) optimization */
5  if (unlikely( REG(EBX) || MEM_E(rbuf(0) + 4) || REG(EDI) || . . . ))
6  {
7      /* propagation body */
8      REG(EBX) = MEM_E(rbuf(0) + 4);
9      . . .
10     REG(EDI) |= MEM_E(rbuf(2) − rbuf(1) −200);
11     REG(ECX) = MEM_E(rbuf(2));
12 }
13
14 /* update the global address array */
15 garg(0) = rbuf(1);
16
17 /* increase index */
18 INC_IDX(3) ;
19
20 /* control transfer */
21 if (likely(IS_VALID_EA(rbuf(0) )) {
22     /* direct jump */
23     goto BB_0xef13a598;
24 } else {
25     if (likely(rbuf(0) == 0xef13a5ba))
26         goto BB_0xef13a5ba;
27     } else {
28         /* hash lookup for computed goto */
29         goto locateBbIdLabel();
30     }
31 }
```

Listing 5.1: Example of secondary code for a BBL.

5.4 ShadowReplica Runtime for DFT operations

The off-line application analysis produces the enqueueing stubs for the primary and the analysis body for the secondary. These are compiled into a single dynamic shared object (DSO) and loaded by ShadowReplica at start up. In this section, we describe various aspects of the runtime environment.

5.4.1 Shadow Memory Management

The shadow memory structure plays a crucial role in the performance of the secondary. Avoiding conditional statements in in-line DFT systems, frequently restricts them to flat memory structures like bitmaps, where reading and updating can be done directly. An approach that does not scale on 64-bit architectures, because the bitmap becomes excessively large. Due to our design, conditional statements do not have such negative performance effects.

We employed the tagmap (shadow memory) structure of libdft-byte that implements byte-per-byte allocation, with a low-cost shadow address translation, and dynamic allocation to achieve a small memory footprint. Shadow memory is maintained in sync with the memory used by the application by intercepting system calls that manage memory (e.g., `mmap()`, `munmap()`, `brk()`) and enqueueing special control commands that allocate and deallocate shadow memory areas. The information sent includes the command code and an address range tuple *(offset, length)*.

To access shadow memory, we first consult a page table like data structure, which points to a per-page flat structure. The mechanism does not require a check for page validity because intercepting the memory management calls ensures it. Accesses to unallocated space are trapped to detect invalid memory addresses.

5.4.2 DFT Sources and Sinks for Taint Analysis

Tagging memory areas, when data are read from a file or the network, is performed by generating code that enqueues another control command in the ring buffer. The data sent to the secondary include the command code, an address range tuple *(offset, length)*, and the tag value (or label). The value with which to tag the address range in shadow memory depends on the particular DFT-logic. For instance, DTA uses binary tags, data can tainted or clean. In this case, we can simply omit the

tag value.

Checking for tagged data can be either done by enqueueing a control command, or it can be done entirely by the secondary, as it is in the case of DTA. For instance, whenever a BBL ends with a `ret` instruction, the secondary checks the location of the return address in the stack to ensure it is clean. If we wanted to check for sensitive information leaks, we would have to issue a command from the primary whenever a `write()` or `send()` system call is made. Note that all checks are applied on the secondary, which can take action depending on their outcome. For DTA, if a check fails, we immediately terminate the application, as it signifies that the application's control flow has been compromised. Other actions could involve logging memory contents to analyze the attack [Yin *et al.*, 2007], or allowing the attack to continue for forensics purposes [Portokalidis *et al.*, 2006].

5.4.3 Ring Buffer

We implemented the ring buffer, as a lock-free, ring buffer structure for a single producer (primary), single consumer (secondary) model. Lamport et al. [Lamport, 1983] initially proposed a lock-free design for the model, however, the design suffers from severe performance overheads on multi-core CPUs due to poor cache performance [Ha *et al.*, 2009]. The most serious problem we observed was *forced* L1 eviction that occurs as the secondary attempts to read the page that was just modified by the primary, while it is still in the primary's L1 cache and before it is committed to RAM. In certain cases, the overhead due to ring-buffer communication increased to the point that our framework became unusable. To mitigate the problem, we switched to a N-way buffering scheme [Zhao *et al.*, 2008], where the ring-buffer is divided into N separate sub-buffers, and primary and secondary threads work on different sub-buffers at any point of time, maintaining a distance of least as much as the size of a single sub-buffer.

5.4.4 Data Encoding

Data enqueued in the ring buffer belong to one of the following three groups: (a) effective addresses (EAs), (b) BBIDs, and (c) control commands. The first two groups are four-bytes long, or the same as the address size if you prefer. Control commands are normally longer, with the four-bytes indicating the command, and a variable number of arguments following. We exploit the fact that valid EAs point to user-space ranges (0G $\sim$ 3G range on x86 Linux) to quickly differentiate them from

other entries without using additional bits. We map BBIDs and control commands to address ranges that correspond to kernel space (3G $\sim$ 4G range on x86 Linux), which immediately differentiates them from EAs that are always in lower address ranges. This design is easily applicable on 64-bit architectures and other operating systems.

5.5 ShadowReplica Evaluation

We evaluated ShadowReplica using common command-line utilities, the SPEC CPU2006 benchmark suite, and a popular web and database (DB) server. Our aim is to quantify the performance gains of our system, and analyze the effectiveness of the various optimizations and design decisions we took. We close this section with an effectiveness (security) evaluation of the DTA tool we built over ShadowReplica.

5.5.1 Effectiveness of Optimizations

Table 5.1 summarizes the effects our optimizations had on reducing the amount of information that needs to be communicated to the secondary, and the effectiveness of the ring buffer fast checking optimization. The *# BBLs* column indicates the number of distinct BBLs discovered for each application, using the profiling process outlined in Sec. 5.3.1. *# Ins.* gives the average number of instructions per BBL. The *Unopt* column shows the average number of enqueue operations to the ring buffer that a naive implementation would require per BBL. The *Intra*, *DFT*, *Inter*, and *Exec* columns correspond to the number of enqueue operations after progressively applying the intra-block, DFT, and inter-block optimizations presented in Sec. 5.3.2.1, as well as the optimizations related to control flow recording (Sec. 5.3.2.2). *Rbuf* shows the percentage of BBLs that need to be instrumented with checks for testing if the ring buffer is full, according to the fast checking algorithm (Sec. 5.3.2.3). *# DFT operands*, shows *per BBL* average number of register and memory operands appeared from the secondary's analysis body. The last column, *Time* shows the time for offline analysis to generate codes for the primary and the secondary.

On average, a naive implementation would require 4.17 enqueue operations per BBL, for communicating EAs and control flow information to the secondary, and instrument 100% of the BBLs with code that checks for ring buffer overflows. Our optimizations reduce the number of enqueueing

Category	Application	# BBLs	# Ins.	Unopt	Optimization				Rbuf	# DFT operands	Time (sec.)
					Intra	DFT	Inter	Exec			
Utilities	bzip2	6175	8.59	4.97	2.93	2.62	2.21	1.40	39.70 %	2.69	156.66
	tar	8615	5.20	2.31	1.82	1.70	1.53	0.69	33.49 %	4.52	141.79
SPEC2006	473.astar	5069	7.38	4.09	2.48	2.15	2.05	1.40	22.11 %	3.30	89.78
	403.gcc	78009	4.37	2.54	1.93	1.84	1.42	0.55	56.87 %	5.57	8341.53
	401.bzip2	4588	7.85	4.12	2.75	2.28	2.10	1.27	32.96 %	2.77	109.60
	445.gobmk	25939	5.97	3.04	2.11	1.90	1.72	0.95	27.09 %	3.56	674.74
	464.h264	11303	5.76	4.88	3.12	3.04	1.72	0.84	74.78 %	3.57	242.67
	456.hmmer	7212	19.14	12.62	6.99	6.66	6.34	5.95	59.62 %	6.64	115.81
	462.libquantum	3522	9.16	4.66	2.61	2.27	2.04	1.09	56.69 %	3.88	58.09
	429.mcf	3752	5.96	3.21	2.12	1.83	1.76	1.06	26.38 %	3.19	57.62
	471.omnetpp	15209	5.36	2.88	2.11	1.95	1.80	1.05	6.59 %	3.31	234.35
	400.perl	27391	5.71	2.86	1.89	1.82	1.64	0.84	7.79 %	4.47	421.10
	458.sjeng	5535	5.29	2.49	1.74	1.65	1.54	0.82	21.51 %	4.07	81.51
	483.xalanc	34881	4.79	2.02	1.54	1.44	1.34	0.50	21.40 %	5.97	1189.45
Server application	Apache	23079	8.39	3.17	1.91	1.84	1.6	1.05	28.08 %	5.57	706.63
	MySQL	40393	6.87	4.21	2.19	2.08	1.80	1.10	11.00 %	5.42	1486.60
Average		18792	7.24	4.17	2.52	2.32	2.04	1.29	32.88 %	4.28	881.74

Table 5.1: Results from ShadowReplica static analysis.

operations to a mere 1.29 per BBL, while only 32.88% of BBLs are instrumented to check for ring buffer overflows. A reduction of 67.12%.

While offline analysis requires a small amount of time (on average 881 sec.) for most programs, 403.gcc takes exceptionally long (8341 sec.) to complete. This is due to the program structure, which has a particularly dense CFG, thus making the *Rbuf* optimization wasting a long time ($\sim$ 6500 sec.) only to enumerate all available cycles to perform the analysis described in Sec. 5.3.2.3. We believe that this can be alleviated by parallelizing the algorithm.

To evaluate the impact of our various optimizations on the primary's performance, we used two commonly used Unix utilities, `tar` and `bzip2`. We selected these two because they represent different workloads. `tar` performs mostly I/O, while `bzip2` is CPU-bound. We run the GNU versions of the utilities natively and over ShadowReplica, progressively enabling our optimizations against a Linux kernel source "tarball" (v3.7.10; $\sim$476MB).

We measured their execution time with the Unix `time` utility and draw the obtained results in

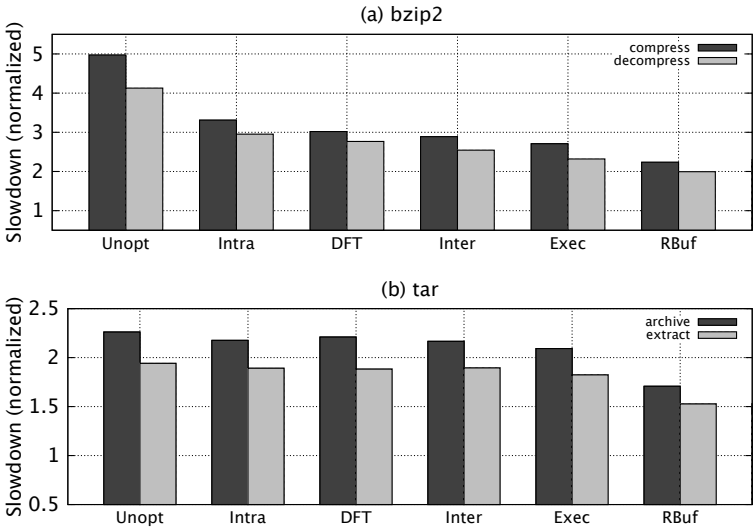


Fig. 5.5: The slowdown of the primary process imposed by ShadowReplica, and the effects of our optimizations, when running `bzip2` and `tar`.

Fig. 5.5. *Unopt* corresponds to the runtime performance of an unoptimized, naive, implementation, whereas *Intra*, *DFT*, *Inter*, *Exec*, and *RBuf* demonstrate the benefits of each optimization scheme, when applied cumulatively. With all optimizations enabled, the slowdown imposed to `bzip2` drops from 5×/4.13× down to 2.23×/1.99× for compress/decompress (55%/51.82% reduction). Similarly, `tar` goes from 2.26×/1.94× down to 1.71×/1.53× for archive/extract (24.33%/ 21.13% reduction). It comes as no surprise that I/O bound workloads, which generally also suffer smaller overheads when running with in-lined DFT, benefit less from ShadowReplica. We also notice that *Intra* and *RBuf* optimizations have larger impact on performance.

5.5.2 ShadowReplica Performance

SPEC CPU2006 Fig. 5.6 shows the overhead of running the SPEC CPU2006 benchmark suite under ShadowReplica, when *all* optimizations are enabled. *Primary* corresponds to the slowdown imposed by the primary thread alone. *Primary+Secondary* is the overhead of ShadowReplica when both the primary and secondary threads and running, and the secondary performs full-fledged DFT. Finally, *In-line* denotes the slowdown imposed when the DFT process executes in-line with the application, under our accelerated DFT implementation with TFA optimizations. On average, *Primary* imposes a 2.72× slowdown on the suite, while the overhead of the full scheme

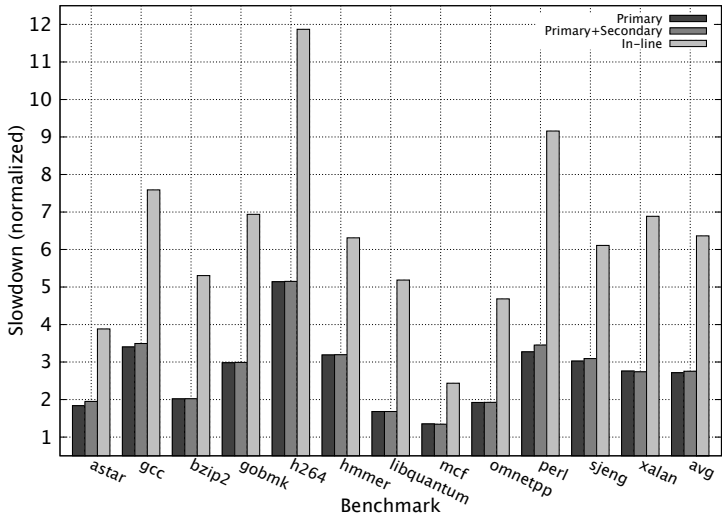


Fig. 5.6: Running the SPEC CPU2006 benchmark suite with ShadowReplica and TFA optimizations.

(*Primary+Secondary*) is 2.75×. *In-line* exhibits a 6.37× slowdown, indicating the benefits from the decoupled and parallelized execution of the DFT code (56.82% reduction on the performance penalty).

During our evaluation, we noticed that for some benchmarks (astar, perlbmk, gcc, sjeng) the slowdown was not bound to the primary, but to the secondary. These programs generally required a high number of DFT operands to be sent to the secondary (*# DFT operands* from Table 5.1) compared to the number of enqueued entries.

Apache and MySQL We proceed to evaluate ShadowReplica with larger and more complex software. Specifically, we investigate how ShadowReplica behaves when instrumenting the commonly used Apache web server. We used Apache v2.2.24 and let all options to their default setting. We measured Apache's performance using its own utility `ab` and static HTML files of different size. In particular, we chose files with sizes of 1KB, 10KB, 100KB, and 1MB, and run the server natively and with ShadowReplica with all optimizations enabled. We also tested Apache with and without SSL/TLS encryption (i.e., *SSL* and *Plaintext*, respectively). Fig. 5.7 (a) illustrates our results. Overall, ShadowReplica has a 24% performance impact, on average, when Apache is serving files in plaintext. Unsurprisingly, the slowdown is larger when running on top of SSL (62%). The reason behind this behavior is that the intensive cryptographic operations performed by SSL make the

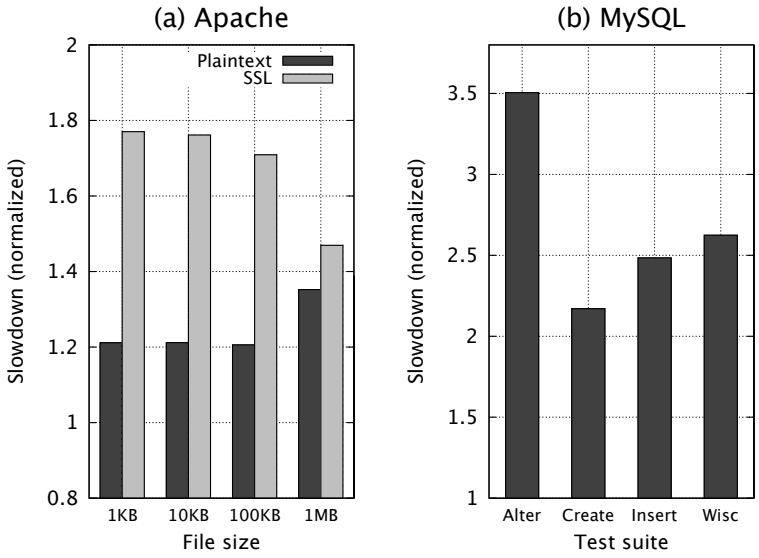


Fig. 5.7: The slowdown incurred by ShadowReplica on the Apache web server and MySQL DB server.

server CPU-bound. In Fig. 5.7 (b), we present similar results from evaluating with the MySQL DB server, a multi-threaded application which spawned $10 \sim 20$ threads during its evaluation. We used MySQL v5.0.51b and its own benchmark suite (`sql-bench`), which consists of four different tests that assess the completion time of various DB operations like table creation and modification, data selection and insertion, and so on. The average slowdown measured was $3.02\times$ ($2.17\times - 3.5\times$).

5.5.3 Computational Efficiency of ShadowReplica

One of the goals of ShadowReplica was to also make DFT more efficient computation-wise. In other words, we not only desired to accelerate DFT, but also do it using less CPU resources. To evaluate this aspect of our approach, we chose two benchmarks from the SPEC CPU2006 suite; 401.bzip2 and 400.perl. Our choice was was not arbitrary. During our performance evaluation, we observed that in the first benchmark, DFT was running faster than the application. Performance is, hence, bound by the primary thread. On the other hand, in the second benchmark the secondary thread, performing DFT, was slower, hence its performance is bound by the secondary thread.

We run ShadowReplica and our accelerated in-line DFT implementation with these two benchmarks, and measured their CPU usage using the `perf` tool. Fig. 5.8 presents the results of our

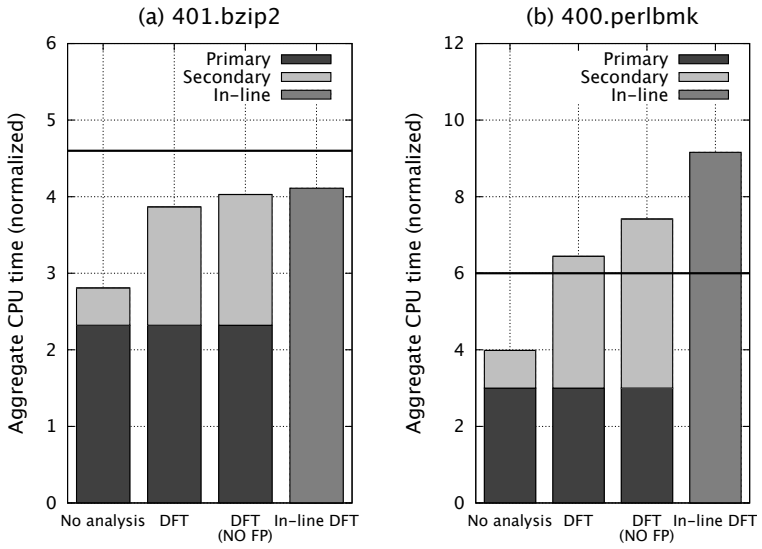


Fig. 5.8: Aggregated CPU time consumed by two SPEC CPU2006 benchmarks when run under different configurations of ShadowReplica, and under in-line DFT (TFA).

experiment. We run ShadowReplica with both primary and secondary threads running, having the secondary perform no analysis (*No analysis*), implementing DFT using all optimizations (*DFT*), and without the FastPath optimization from LIFT [Qin *et al.*, 2006] (*DFT (NO FP)*). The last column (*in-line DFT*) shows the result for a DFT implementation with TFA optimizations. CPU usage is partitioned to show the amount of CPU cycles taken from the primary and secondary threads separately. The darker horizontal line visualizes the tipping point where the secondary thread starts dominating performance (i.e., it is slower than the primary), when we are running ShadowReplica with DFT and all optimizations enabled.

A take-out from these results is that the aggregated CPU usage of ShadowReplica is less or equal than that of in-line DFT (TFA). Astoundingly, in the case of 401.bzip2, we are so much more efficient that we require $\sim$30% less CPU cycles to apply DFT.

5.5.4 Security

The purpose of developing the DTA over ShadowReplica was not to provide solid solutions, but to demonstrate that our system can indeed facilitate otherwise complex security tools. Nevertheless, we tested their effectiveness using the same set of exploits that we used for libdft listed from

Table 3.1. In all cases, we were able to successfully prevent the exploitation of the corresponding application. During the evaluation, DTA did not generate any false positives, achieving the same level of correctness guarantees to our previous implementation of libdft and TFA. Having DTA implemented mostly from the secondary, performance overhead was negligible by having $\sim 5\%$ slowdown.

5.6 Generalization of ShadowReplica Approach

The main motivation behind ShadowReplica has been to accelerate security techniques with well established benefits, by providing a methodology and framework that can utilize the parallelism offered by multi-core CPUs. Besides DFT and DTA, we also implemented a control-flow integrity [Abadi *et al.*, 2005] tool to demonstrate the flexibility of our approach. We argue that many other analyses can benefit from it.

Control-flow Integrity. CFI [Abadi *et al.*, 2005], similarly to DTA, aims at preventing attackers from compromising applications by hijacking their control flow. Programs normally follow predetermined execution paths. CFI enforces program flow to follow one of these predetermined paths. Determining a complete CFG of a program is a challenging task in itself, but assuming such a CFG is available, CFI operates as follows. Run-time checks are injected in the application to ensure that control flow remains within the CFG. These checks are placed before every indirect control flow transfer in the application and check that the destination is one of the intended ones. Basically, all possible destinations of a particular control flow instruction are assigned the same id, which is validated before transferring control. While this can be overly permissive, because multiple instructions may have the same targets assigning the same id to the super-set of destinations, this approach allows for fast checks. We implemented CFI with ShadowReplica by using the CFG information extracted during the application profiling, and by generating analysis code that checks whether a control-flow transfer is allowable, using the same control flow enqueueing stubs we used for DFT.

Other Analyses. Dynamic analyses that use DBI frameworks [Luk *et al.*, 2005; Bruening, 2004; Nethercote, 2004] can also readily make use of ShadowReplica (e.g., techniques that focus on memory integrity detection). Valgrind's Memcheck [Nethercote, 2004] uses shadow memory to keep track of uninitialized values in memory and identify their (dangerous) use; Memcheck is an ideal

candidate for accelerating through ShadowReplica. Dr. Memory [Bruening and Zhao, 2011] discovers similar types of programming errors including memory leaks, heap overflows, etc Software-based fault isolation [Wahbe *et al.*, 1993] mechanisms can also be easily supported through our framework by using the existing code analysis for the primary and small modifications to the secondary. Approaches that do not depend on shadow memory can also be supported with moderate engineering effort. Examples include call graph profiling, method counting, and path profiling. Finally, ShadowReplica can be extended to work for analyses that refer to memory contents such as integer overflow detection. Note that analyses of this type are less common for binaries, as they require access to source code.

Part IV

Static IFT

Chapter 6

Integer Error Detection with IntFlow

6.1 IFT for Integer Errors

Integer overflow and underflow, signedness conversion, and other types of arithmetic errors in C/C++ programs are among the most common software flaws that result in exploitable vulnerabilities. Despite significant advances in automating the detection of arithmetic errors, existing tools have not seen widespread adoption mainly due to their increased number of false positives. Developers rely on wrap-around counters, bit shifts, and other language constructs for performance optimizations and code compactness, but those same constructs, along with incorrect assumptions and conditions of undefined behavior, are often the main cause of severe vulnerabilities. Accurate differentiation between legitimate and erroneous uses of arithmetic language intricacies thus remains an open problem.

As a step towards addressing these issues, we propose an approach that combines static code instrumentation with Information Flow Tracking to improve the accuracy of arithmetic error detection, focusing on reducing the number of false positives, i.e.,, developer-intended code constructs that violate language standards. Our tool, IntFlow, uses information flow tracking to reason about the severity of arithmetic errors by analyzing the information flows related to them that lead to sensitive sinks (sensitive integer operations that are likely to demonstrate undefined behavior) and, depending on where each flow originates from, flags them as suspicious or benign. The main intuition behind this approach is that arithmetic errors are critical when *(a)* they are triggered by or depend on values originating from untrusted locations, or *(b)* a value affected by an arithmetic error propagates

to sensitive locations, such as the arguments to functions like `malloc()` and `strcpy()`. With IntFlow, we aim to reduce the number of false positives produced by other static arithmetic error checkers [Dietz *et al.*, 2012].

While libdft, our baseline DFT tool, employs dynamic DFT approach and implement DTA tool as its primary application, IntFlow differs from it in two ways. First, it applies IFT to new problem domain of the integer errors. IntFlow uses IFT to mitigate the issue of too many false reportings. Second, it statically applies IFT leveraging LLVM compiler framework [Lattner and Adve, 2004] to analyze program source at compile time. This discloses type and signedness information crucial in handling integer errors. It also incurs less runtime overhead even when it is compared to highly optimized dynamic DFT solutions. (TFA, ShadowReplica).

To demonstrate the effectiveness of our approach, we evaluated IntFlow with real world programs and vulnerabilities and verified that it successfully identifies all the real world vulnerabilities for the applications of our testbed, generating 89% less false positives compared to IOC [Dietz *et al.*, 2012], our reference arithmetic error detection tool.

6.2 IntFlow Overview

The security community is still unsuccessful in completely eliminating the problem of integer errors even after years of effort [Dietz *et al.*, 2012; Molnar *et al.*, 2009; Zhang *et al.*, 2010; Wang *et al.*, 2012]. One of the main reasons is the difficulty in distinguishing *critical* errors, which may lead to reliability issues or security flaws, from uses of undefined constructs stemming from certain programming practices. The latter are regarded as errors by rigorous static checkers like IOC, as they strictly follow the language standard and report all violations found. In this work, we attempt to pinpoint critical, possibly exploitable arithmetic errors among all arithmetic violations, which include numerous less critical and often developer-intended uses of undefined behavior. Although many programming choices deviate from the language specification, IntFlow examines the conditions under which such constructs signify critical bugs, by focusing only on detecting errors that might break the functionality of the program or lead to a security flaw.

Before describing IntFlow's design, we first provide a concrete definition of what we consider critical arithmetic errors.

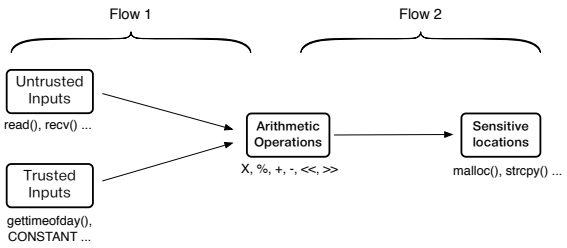


Fig. 6.1: Information flows to and from the location of an arithmetic error.

Definition 1. *An arithmetic error is potentially critical if it satisfies any of the following conditions:*

1. *At least one of the operands in an erroneous arithmetic operation originates from an* ***untrusted source****.*

2. *The result of an erroneous operation propagates to a* ***sensitive program location****.*

As capturing the intention of developers is a hard problem, IntFlow focuses on the detection of arithmetic errors that might constitute exploitable vulnerabilities or cause reliability issues. This is achieved not only by focusing on the identification of violations according to the language standard, which in itself is a tractable problem, but also by considering the information flows that affect the erroneous code. The rationale behind this definition is that *(a)* arithmetic errors influenced by external and potentially untrusted sources, such as sockets, files, and environment variables, may be exploited through carefully crafted inputs, and *(b)* arithmetic errors typically result in severe vulnerabilities when they affect sensitive library and system operations, such as memory allocation and string handling functions.

The two conditions of Definition 1 are reflected in IntFlow by two different modes of operation, *blacklisting* and *sensitive* (in addition to a third *whitelisting* mode), discussed in Section 6.3.3. Although the existence of either condition is an indication of a critical error, arithmetic violations for which both conditions hold are more severe, as they can potentially allow input from untrusted sources to misuse critical system functions—IntFlow's different modes of operation can be combined to detect such errors.

Figure 6.1 visualizes the definition with different types of information flows that may involve erroneous arithmetic operations. Critical errors are related to information flows that originate from untrusted inputs, or that eventually reach sensitive operations, such as system calls, through value

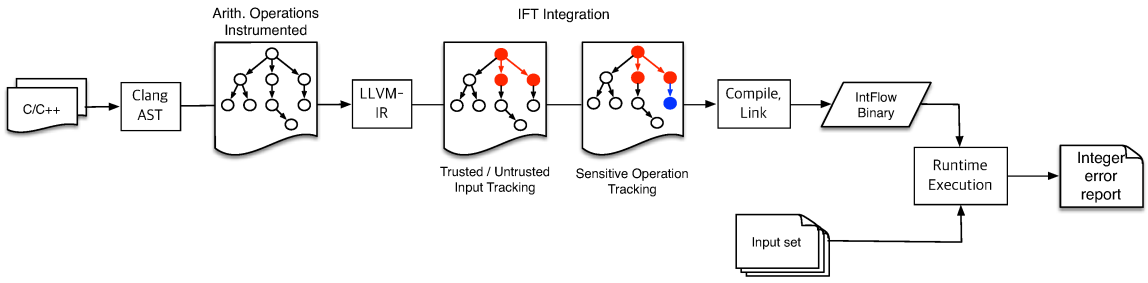


Fig. 6.2: Overall architecture of IntFlow.

propagation. In cases where the input of an arithmetic operation is untrusted (Definition 1.1) or a sensitive sink is reached (Definition 1.2), the error is flagged as critical. In contrast, arithmetic errors influenced only by benign inputs are considered less likely to be used in exploitation attempts.

6.3 IntFlow Design and Implementation

In this section we present the design and implementation of IntFlow, a tool that combines Information Flow Tracking (IFT) [Moore, 2013] with a popular integer error checking tool [Dietz *et al.*, 2012] to improve the accuracy of arithmetic error detection. The main goal of IntFlow is to reduce the number of false positives produced by other static arithmetic error checkers. In this context, the term "false positive" refers to reporting developer-intended violations as critical errors. Although from the perspective of the language standard these correspond to erroneous code, the prevalence of such constructs makes reports of such issues a burden for security analysts, who are interested only in critical errors that may form exploitable vulnerabilities.

6.3.1 Main Components

Integer Overflow Checker operates at the abstract syntax tree (AST) level produced by Clang, a C/C++ front-end of LLVM [Lattner and Adve, 2004]. It instruments all arithmetic operations, such as additions, multiplications and shifts, as well as most of unary, casting, and type conversion operations. In contrast to previous tools that focus on a subset of integer errors (typically overflows and underflows), Integer Overflow Checker provides protection against a broader range of integer errors. Even though it focuses mainly on errors with undefined behavior based on the language standards, it can also protect against errors that do not fall into this category.

Integer Overflow Checker instruments *all* arithmetic operations that may lead to an erroneous result, and inserts checks accordingly. Essentially, for each integer operation inside a basic block, additional basic blocks that implement the error-checking logic are added and users are allowed to register callback functions for error handling. Similarly to other integer error detection systems, the fact that Integer Overflow Checker instruments blindly all arithmetic operations is a major source of false positives as well as the significant overhead, while IntFlow's active provisioning allows it to reduce false positives by eliminating checks for non-critical violations. Integer Overflow Checker is a major component of our architecture, as it provides assurance that all potentially serious arithmetic errors can be checked. It is then up to the information flow analysis to identify and report only the critical ones.

For IntFlow's information tracking mechanism we employ `llvm-deps` [Moore, 2013], an LLVM compiler pass implementing static Information Flow Tracking in a manner similar to classic data flow analysis [Aho *et al.*, 2006]. It is designed as a context sensitive (inter-procedural) analysis tool that allows forward and backward slicing on source and sink pairs of our choice using the DSA [Lattner *et al.*, 2007] algorithm. DSA performs context-sensitive, unification-based points-to analysis that allows us to track data flows among variables referred by pointer aliases. It is important to note that the analysis scope of `llvm-deps` is limited to a single object file, as it is implemented as a compile-time optimization pass and not as a link-time optimization pass. Finally, due to the use of `llvm-deps`, IntFlow only examines explicit flows during its IFT analysis and ignores possible implicit flows.

6.3.2 Putting It All Together

Figure 6.2 illustrates the overall architecture of IntFlow: Integer Overflow Checker adds checks to the integer operations that are exposed by Clang in the AST and then `llvm-deps` performs static IFT analysis on the LLVM intermediate representation (IR). To reduce unnecessary checks that may lead to false positives, IntFlow uses `llvm-deps` to examine only certain flows of interest. As discussed in the previous section, IntFlow examines only flows stemming from untrusted sources, or ending to sensitive sinks. Initially, IntFlow performs *forward* slicing: starting from a particular source used in a potentially erroneous arithmetic operation, it examines whether the result of the operation flows into sinks of interest. Once such a source is found, IntFlow performs *backward*

slicing, to verify that the sink is actually affected by it. Since the flow tracking mechanism does not offer full code coverage, we employ this two-step process to gain confidence on the accuracy of the flow and verify its validity. Once the source is reached when using backward slicing starting from the sink, the flow is considered established.

After compiling and linking the IR, the resulting binary is exercised to identify critical errors, since the error checking mechanism triggers dynamically. Developers should execute the augmented binary with a broad range of inputs to exercise as many execution paths as possible, and identify whether they cause critical errors that can potentially lead to exploitable vulnerabilities.

6.3.3 Modes of Operation

As discussed in Section 6.2, IntFlow uses two different types of information flow to pinpoint errors. The first associates untrusted inputs with integer operations while the second associates the result of integer operations with its use in sensitive system functions. Once Integer Overflow Checker inserts checks in all arithmetic operations that may lead to an error, IntFlow eliminates unnecessary checks by operating in one of the following modes:

- In *blacklisting* mode, IntFlow only maintains checks for operations whose operands originate from untrusted sources and removes all other checks.
- In *sensitive* mode, IntFlow only maintains checks for operations whose results may propagate to sensitive sinks.
- In *whitelisting* mode, all checks for operations whose arguments come from trusted sources are removed.

6.3.3.1 Trusted and Untrusted Inputs

For each operation that may result in an arithmetic error, IntFlow's IFT analysis determines the origin of the involved operands. IntFlow then classifies the origin as either trusted or untrusted, and handles it accordingly, using one of the following two modes of operation.

Blacklisting: Input sources that can be affected by external sources are considered untrusted, since carefully crafted inputs may lead to successful exploitation. If any of the operands has a value affected by such a source, IntFlow retains the error checking instrumentation. System and library

calls that read from untrusted sources, such as `read()` and `recv()`, are examples of this type of sources.

Whitelisting: Erroneous arithmetic operations for which all operands originate from trusted sources are unlikely to be exploitable. Thus, for those cases, IntFlow safely removes the error checks inserted by Integer Overflow Checker at the instrumentation phase. Before an operation is verified as safe, IntFlow needs to examine the origin of *all* data flowing to that operation. Values derived from constant assignments or from safe system calls and library functions, e.g., `gettimeofday()` or `uname()`, are typical examples of sources that can be trusted, and thus white-listed.

Following either of the above two approaches, IntFlow selects the unsafe integer operations that will be instrumented with protection checks. These modes of operation can be complemented by IntFlow's third mode, which refines the analysis results for the surviving checks.

6.3.3.2 Sensitive Operations

In this mode, IntFlow reports flows that originate from integer error locations and propagate to sensitive sinks, such as memory-related functions and system calls. Moreover, in contrast to the previous modes, whenever an integer error occurs, the error is not reported at the time of its occurrence, but only once it propagates as input into one of the sensitive sinks. This is very effective in suppressing false positives, since errors that do not flow to a sensitive operation are not generally exploitable.

To report errors at sensitive sink locations, IntFlow performs the following operations:

- Initially, the tool identifies all integer operations whose results may propagate into a sensitive sink at runtime. Any checks that do not lead to sensitive sinks are not exploitable and thus are eliminated. A global array is created for each sensitive sink, holding one entry per arithmetic operation affecting it.

- Whenever an integer operation generates an erroneous result, its respective entries in the affected global arrays are set to `true`. If the operation is reached again but without generating an erroneous result, before reaching a sensitive location, the respective entry is set to `false`, denoting that the result of the sensitive operation will not be affected by this operation.

- If the execution reaches a sensitive function, the respective global array is examined. Execution is interrupted if one or more entries are set to `true`, as an erroneous value from any

previous integer operation may affect its outcome.

Essentially, IntFlow keeps track of all the locations in the code that may introduce errors affecting a sensitive sink at compilation time. Once a sensitive sink is reached during runtime, IntFlow examines whether *any* of the error locations associated with the sink triggered an error in the current execution flow and in that case terminates the program.

6.4 IntFlow Implementation

IntFlow is implemented as an LLVM [Lattner and Adve, 2004] pass written in ∼3,000 lines of C++ code. Briefly, it glues together its two main components (Integer Overflow Checker and `llvm-deps`) and supports fine-tuning of its core engine through custom configuration files.

IntFlow can be invoked by simply passing the appropriate flags to the compiler, without any further action needed from the side of the developer. IntFlow's pass is placed at the earliest stage of the LLVM pass dependency tree to prevent subsequent optimization passes from optimizing away any critical integer operations. During compilation, arithmetic error checks are inserted by Integer Overflow Checker, and then selectively filtered by IntFlow. Subsequent compiler optimizations remove the filtered Integer Overflow Checker basic blocks.

IntFlow offers developers the option to explicitly specify arithmetic operations or sources that need to be whitelisted or blacklisted. In addition, it can be configured to exclude any specific file from its analysis or ignore specific lines of code. Developers can also specify the mode of operation that IntFlow will use, as well as override or extend the default set of sources and sinks that will be considered during information flow analysis. Finally, they can specify particular callback actions that will be triggered upon the discovery of an error, such as activating runtime logging or exiting with a suitable return value. These features offer great flexibility to developers, enabling them to fine-tune the granularity of the generated reports, and adjust the built-in options of IntFlow to the exact characteristics of their source code.

6.5 IntFlow Evaluation

In this section, we present the results of our experimental evaluation using our prototype implementation of IntFlow. To assess the effectiveness and performance of IntFlow, we look into the following aspects:

- What is the accuracy of IntFlow in detecting and preventing critical arithmetic errors?
- How effective is IntFlow in reducing false positives? That is, how good is it in omitting developer-intended violations from the reported results?
- When used as a protection mechanism, what is the runtime overhead of IntFlow compared to native execution?

Our first set of experiments aims to evaluate the IntFlow's ability to identify and mitigate critical errors. For this purpose, we use two datasets consisting of artificial and real-world vulnerabilities. Artificial vulnerabilities were inserted to a set of real-world applications, corresponding to various types of MITRE's Common Weakness Enumeration (CWE) [MIT,]. This dataset provides a broad test suite that contains instances of many different types of arithmetic errors, which enables us to evaluate IntFlow in a well-controlled environment, knowing exactly how many bugs have been inserted, as well as the nature of each bug. Likewise, our real-world vulnerability dataset consists of applications such as image and document processing tools, instant messaging clients, and web browsers, with known CVEs, allowing us to get some insight on how well IntFlow performs against real-world, exploitable bugs.

In our second round of experiments, we evaluate the effectiveness of IntFlow's information flow tracking analysis in reducing false positives, by running IntFlow on the SPEC CPU2000 benchmark suite and comparing its reported errors with those of Integer Overflow Checker. Integer Overflow Checker instruments all arithmetic operations, providing the finest possible granularity for checks. Thus, by comparing the reports produced by IntFlow and Integer Overflow Checker, we obtain a base case for how many non-critical errors are correctly ignored by the IFT engine.

Finally, to obtain an estimate of the tool's runtime overhead, we run IntFlow over a diverse set of applications of varying complexity, and establish a set of performance bounds for different types

of binaries. All experiments were performed on a system with the following characteristics: 2× Intel(R) Xeon(R) X5550 CPU @ 2.67GHz, 2GB RAM, i386 Linux.

6.5.1 Accuracy

In all experiments for evaluating accuracy, we configured IntFlow to operate in *whitelisting* mode, since this mode produces the greatest number of false positives, as it preserves most of the Integer Overflow Checker checks among the three modes. Thus, whitelisting provides us with an estimation of the worst-case performance of IntFlow, since the other two modes perform more fine-tuned instrumentation.

6.5.1.1 Evaluation Using Artificial Vulnerabilities

To evaluate the effectiveness of IntFlow in detecting critical errors of different types, we used seven popular open-source applications with planted vulnerabilities from nine distinct CWE categories.[1] Table 6.1 provides a summary of the applications used and the respective CWEs.

Each application is replicated to create a set of test-case binaries. In every test-case binary—essentially an instance of the real-world application—a vulnerability is planted and then the application is compiled with IntFlow. Subsequently, each test-case binary is executed over a set of benign and malicious inputs (inputs that exploit the vulnerability and result in abnormal behavior). A correct execution is observed when the binary executes normally on benign inputs or terminates before it can be exploited on malicious inputs.

Overall, IntFlow was able to correctly identify 79.30% (429 out of 541) of the planted artificial vulnerabilities. The 20.7% missed are due to the accuracy limitations of the IFT mechanism, which impacts the ability of IntFlow to correctly identify flows, and also due to vulnerabilities triggered by implicit information flows (i.e., non-explicit data flows realized by alternate control paths), which our IFT implementation (`llvm-deps`) is not designed to capture.

Table 6.1: Summary of the applications and CWEs used in the artificial vulnerabilities evaluation.

Applications	Cherokee (1.2.101) Grep (2.14), Nginx (1.2.3), Tcpdump (4.3.0), W3C (5.4.0), Wget (1.14), Zshell (5.0.0)
CWEs	CWE-190 CWE-191, CWE-194, CWE-195, CWE-196, CWE-197, CWE-369, CWE-682, CWE-839

Table 6.2: CVEs examined by IntFlow.

Program	CVE Number	Type	Detected?
Dillo	CVE-2009-3481	Integer Overflow	Yes
GIMP	CVE-2012-3481	Integer Overflow	Yes
Swftools	CVE-2010-1516	Integer Overflow	Yes
Pidgin	CVE-2013-6489	Signedness Error	Yes

6.5.1.2 Mitigation of Real-world Vulnerabilities

In our next experiment, we examined the effectiveness of IntFlow in detecting and reporting real-world vulnerabilities. For this purpose, we used four widely-used applications and analyzed whether IntFlow detects known integer-related CVEs included in these programs. Table 6.2 summarizes our evaluation results. IntFlow successfully detected *all* the exploitable vulnerabilities under examination. From this small-scale experiment, we gain confidence that IntFlow's characteristics are maintained when the tool is applied to real world programs, and therefore it is suitable as a detection tool for real-world applications.

6.5.1.3 False Positives Reduction

Reducing the number of false positives is a major goal of IntFlow, and this section focuses on quantifying how effective this reduction is. For our first measurement we used SPEC CPU2000, a suite that contains C and C++ programs representative of real-world applications. Since Integer Overflow Checker is a core component of IntFlow, we chose to examine the same subset of benchmarks of CPU2000 that was used for the evaluation of Integer Overflow Checker to measure the improve-

[1]The modified applications for this experiment were provided by MITRE for testing the detection of integer error vulnerabilities, as part of the evaluation of a research prototype [Benameur *et al.*, 2013].

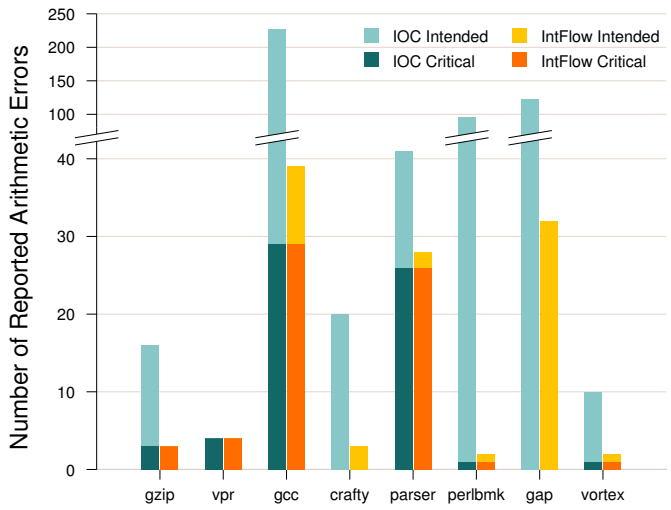


Fig. 6.3: Number of critical and developer-intended arithmetic errors reported by IOC and IntFlow for the SPEC CPU2000 benchmarks.

ments of IntFlow's IFT in comparison to previously reported results [Dietz *et al.*, 2012]. We ran the SPEC benchmarks using the "test" data sets for both Integer Overflow Checker and IntFlow, so that we could manually analyze all the reports produced by IOC and classify them as true or false positives, Once all reports were categorized based on Definition 1, we examined the respective results of IntFlow. We report our findings in Figure 6.3. From the result, reader can observe that IntFlow identifies the same number of critical errors (dark sub-bars), while it reduces significantly the number of reported developer-intended violations.

IntFlow was able to correctly identify all the critical errors (64 out of 64) triggered during execution, and reduced the reports of developer-intended violations by ∼89% (419 out of 471).

6.5.1.4 Real-world Applications

In Section 6.5.1.2 we demonstrated how IntFlow effectively detected known CVEs for a set of real-world applications. Here, we examine the reduction in false positives achieved when using IntFlow's core engine instead of static instrumentation with Integer Overflow Checker alone. To collect error reports, we ran each application with benign inputs as follows: for Gimp, we scaled an image logo and exported it as GIF; for SWFTools, we used the `pdf2swf` utility with a popular e-book as input; for Dillo, we visited a conference webpage and downloaded the list of notable publications; and for

Table 6.3: Number of False Positives reported by IOC and IntFlow for the real-world programs.

	Overall	Dillo	Gimp	Pidgin	SWFTools
Integer Overflow Checker	330	31	231	0	68
IntFlow	82	26	13	0	43

Pidgin, we performed various common tasks, such as registering a new account and logging in and out. Table 6.3 shows the reports generated by Integer Overflow Checker and IntFlow, respectively.

Overall, IntFlow was able to suppress 75% of the errors reported by Integer Overflow Checker during the execution of the applications on benign inputs. Although this evaluation does not provide full coverage on the number of generated reports (for instance, we did not observe any false positives for pidgin with the tests we performed), it allows us to obtain an estimate of how well IntFlow performs in real world scenarios.

As the last part of our false positive reduction, we exercised vanilla versions of each application used in Section 6.5.1.1 over sets of benign inputs and examined the output. Since those inputs produce the expected output, we assume that all reported violations are developer-intended either explicitly or implicitly. With this in mind, we compared the error checks of IntFlow with those of Integer Overflow Checker to quantify the ability of IntFlow in removing unnecessary checks. Overall, IntFlow eliminated 90% of the false checks (583 out of 647) when tested with the default set of safe inputs. This reduction was achieved due to the successful identification of constant assignments and the whitelisting of secure system calls, as discussed in Section 6.2.

It should be noted that the effectiveness in the reduction of false positives is highly dependent on the nature of each application, as well as on the level of the execution's source coverage. That is, the more integer operations occur throughout the execution, the greater the expected number of false positives. For instance, Gimp's functionality is tightly bound to performing arithmetic operations for a number of image processing actions, and thus Integer Overflow Checker reports many errors, most of which are developer-intended, while Dillo does not share the same characteristics and as a result exhibits a smaller reduction in false positives.

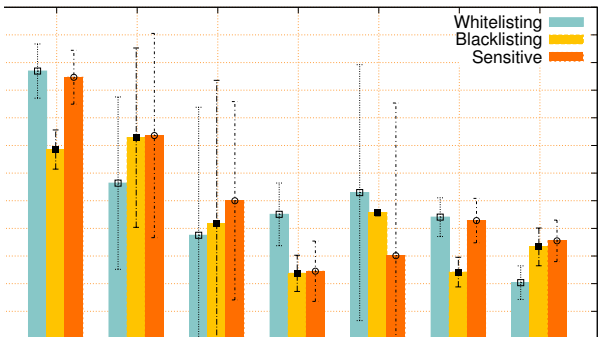


Fig. 6.4: Runtime overhead for the real-world applications (normalized over native execution).

6.5.2 Runtime Overhead

Although IntFlow was not designed as a runtime detection tool but rather as an offline integer error detection mechanism, one may wonder whether it could be customized to offer runtime detection capabilities. In this section, we seek to examine the performance of IntFlow for various applications, when running them with all the automatically inserted arithmetic error checks. For this purpose, we perform a set of timing measurements on the applications used in Section 6.5.1.1. For each run, we measured the time that was required to complete a series of tasks for each of IntFlow's modes of operation, and then normalized the running time with respect to the runtime of the native binary. Reported results are mean values over ten repetitions of each experiment, while the reported confidence intervals correspond to 95%. We ran all binaries natively, and measured user time with the `time` utility.

For `grep`, we search for strings that match a complex regular expression in a 1.2GB file. For `wget`, we download a 1GB file over a 1Gbit/s link from a remote server. For `zshx`, we execute a series of shell commands, and for `tcpdump` we examine packets from a 5.8GB pcap file. For the web servers, Cherokee was configured for IPv4 only, while for Nginx all configurations options were left to their default setting. We measured performance using Apache's `ab` benchmarking utility and static HTML files.

Figure 6.4 shows the results of overhead evaluation. IntFlow incurs high overhead in appli-

cations that involve a large number of integer operations, such as `grep`.[2] We also notice high performance deviation for `wget`, `wwwx`, and `tcpdump`, as they are I/O bound. Although in such applications the overhead is rather prohibitive, and cancels out the benefits of using a static mechanism, in other cases, such as for the `cherokee` and `nginx` servers, the overhead is within an acceptable 20%. Thus, it could be the case that IntFlow might be used as a runtime defense for certain types of applications, i.e., I/O-bound. As each of IntFlow's modes of operation targets different flows and can be fine-tuned by developers, customization can result in different overheads, as different flows dominate the execution of different applications. This is the reason we observe different slowdowns per mode: depending on whether the dominating flows involve sensitive calls (e.g., web servers), the sensitive mode will be slower or faster than the other two modes, and so on.

6.5.3 IntFlow Limitations

Although IntFlow incurs less overhead by being static IFT solution, it also confirms that the approach discloses more accuracy issues than dynamic approach. Firstly, due to limitation of its static IFT component [Moore, 2013], IntFlow's inter-procedural dataflow analysis cannot be extended beyond a certain number of function calls. It is a tunable and we usually set two function distances for this parameter. Secondly, DSA [Lattner *et al.*, 2007] is choice for point-to analysis relate two variables and define dataflow. Although DSA is the most widely adopted analysis of the kind, it is still generates inaccuracy errors. IntFlow therefore is unable to detect every information flows that would lead to false reporting of integer errors requiring user interventions. In other words, it can reduce the number of false reporting but not capable of detecting all incorrect reporting. To complement this limitation we take manual whitelisting approach to label still benign but reported by IntFlow as an integer error.

[2]Based on our experience with the SPEC CPU2000 benchmarks, the overhead on benchmarks with very frequent integer operations, such as `gzip`, is in the range of $\sim$x10, prohibiting IntFlow from being used as a generic runtime detection mechanism for such applications.

Part V

DFT Accuracy Measurement Study

Chapter 7

Measuring DFT accuracy with TaintMark

From this chapter, we discuss about our methodology to evaluate the accuracy of DFT systems. Based on the methodology, we build TaintMark, DFT evalution tool for Android Framework.

7.1 TaintMark Overview

While the research community share the overhead challenge have put efforts in developing practical DFT systems with good performance and scalability, the issue of accurate tracking is rather overlooked. We believe that a core reason for the absence of accuracy evaluations in DFT literature is the lack of an established methodology for understanding accuracy, as well as tools to facilitate its adoption. Indeed, evaluating accuracy is challenging, because it requires ground truth for information flows. While ground truth can be established for simple test benchmarks, Increased visibility into accuracy of DFT systems – as we believe is necessary for practical DFT design – requires testing the DFT technologies on real applications, whose true information flows are unknown.

To address this challenge, we have developed *TaintMark*, a new methodology and tool for understanding accuracy of DFT systems with real applications. TaintMark uses blackbox differential testing to find likely inaccuracies (incomplete or imprecise tracking) in a specified DFT, such as TaintDroid or any other DFTs. Intuitively, TaintMark works as follows: it feeds the application with differentiated, tainted inputs, and looks at the outputs to assess whether they are affected by

Table 7.1: DFT Systems survey.

DFT System			Sources of Inaccuracy		Evaluation	
Name	**Description**	**Layer**	**False Positives**	**False Negatives**	**Performance**	**Accuracy**
TaintDroid [Enck *et al.*, 2010]	Android DFT, trades accuracy for performance (ARM)	interpreter	coarse taints for arrays, IPC, files	interpreter only, implicit flows	Yes	No
TaintDroid-IPC [Enck *et al.*, 2014]	TaintDroid-based, fine IPC taints (ARM)	interpreter	coarse taints for arrays, files	interpreter only, implicit flows	Yes	No
Pebbles TaintDroid [Spahn *et al.*, 2014]	TaintDroid-based, fine array taints (ARM)	interpreter	coarse taints for IPC, files	interpreter only, implicit flows	Yes	No
T-BayesDroid [Tripp and Rubin, 2014]	TaintDroid for propagation, classification for accuracy(ARM)	interpreter	coarse taints for array, IPC, files	interpreter only, implicit flows	Yes	Toy apps
libdft [Kemerlis *et al.*, 2012]	Generic DFT framework, uses Pin for DBI (x86)	binary	no kernel instrumentation	no control flow	Yes	No
Raksha [Dalton *et al.*, 2007]	Full system DFT, HW support for tag propagation (SPARC)	hardware	4-byte granularity	no control flow	Yes	No
Argos [Portokalidis *et al.*, 2006]	Full system DFT, used for signature generation (x86)	emulator	1 tag per register	no control flow	Yes	No
LIFT [Qin *et al.*, 2006]	Windows DFT, disable tracking when not needed (x86-64)	binary	no kernel instrumentation	no file taints	Yes	No
Transformer [Xu *et al.*, 2006]	Transforms C-written interpreters to add DFT	compiler	Points-to inaccuracy	unknown external libraries	Yes	Toy apps

the tainted inputs. If the outputs' values are affected by changes in tainted inputs but the tested DFT does not propagate their taints, then TaintMark will report the test case as a false negative (incompleteness bug). If the outputs' values are not affected by changes in the tainted inputs but the tested taint tracking system does propagate their taints, then TaintMark will report the test case as a false positive (imprecision bug). Numerous additional complexities arise when applying this methodology to real applications, and TaintMark addresses them to develop the first practical accuracy testing tool for DFTs.

7.2 TaintMark Background

Our goal is to develop a methodology to measure the accuracy of DFT system, plus the tool to support it, to start to mend a significant gap in the DFT literature: the absence of understanding accuracy of DFT systems.

7.2.1 DFT Summary

DFT systems track information flows in a computer system from a set of designated *sources* (such as an input device) to a set of designated *sinks* (such as files or network connections). DFT systems can be of two kinds: *static* or *dynamic*. Static DFTs analyze the code to report information flows. Dynamic DFTs track information flows at runtime. In this paper, we focus exclusively on dynamic DFT systems (DFTs for short).

A common method for tracking information flows in DFTs is to associate a *tag* with each memory object, which accumulates identifiers for the designated sources (a.k.a., *taints*). As a program executes on objects in memory, the taints from input objects are propagated into the taint tags of the results. Taint tags can be associated with memory objects at multiple granularities, e.g.,: one tag per bit [Henderson *et al.*, 2014], byte, memory page, primitive type in a language (e.g., ints, floats, primitive arrays, pointers) [Enck *et al.*, 2010]. Taints are often not only propagated between main memory locations but also when writing data to disk. Varied tagging (a.k.a., tainting) granularities apply in these situations, as well: one taint per byte, file, directory, etc. Granularity of tainting creates a generally acknowledged tradeoff between the DFT's performance and memory overheads on one hand and its precision on the other hand. Generally, the finer the granularity, the more memory is consumed and the less efficient the computations are, but the DFT is more precise at identifying true flows. For example, reported performance overheads are around 20× [Henderson *et al.*, 2014] for bit-level tainting and 5× for byte-level tainting [Kemerlis *et al.*, 2012].

DFTs can be applied at multiple levels of abstraction, including full system, executable, or interpreter. The level of tracking creates a generally acknowledged tradeoff between the DFT system's performance and its completeness. For example, tracking taints at the level of an interpreter is generally faster than tracking taints at executable level, but can miss information flows if an interpreted program switches to execute native code. Full-system tainting, which provides the most complete solution, generally requires special hardware to be practical.

Finally, a number of other accuracy tradeoffs are known. For example, reference tainting is known to create explosions (massive false positives) [Slowinska and Bos, 2009], but avoiding reference tainting may result in incomplete tracking (false negatives). As another example, detection of implicit flows can create a precision/completeness tradeoff.

All of these tradeoffs are rules of thumb, as different systems account for tradeoffs differently.

It is therefore unclear exactly how the tradeoffs affect each design, hence case-by-case studies of accuracy are crucial. Yet, such studies are rare, as shown next.

7.2.2 DFT Survey

Table 7.1 shows which of the potential inaccuracy sources outlined in the preceding section apply to various DFTs from prior literature. We survey various DFTs from prior literature [Enck *et al.*, 2010; Enck *et al.*, 2014; Tripp and Rubin, 2014; Kemerlis *et al.*, 2012; Dalton *et al.*, 2007; Portokalidis *et al.*, 2006; Qin *et al.*, 2006; Xu *et al.*, 2006] and observe that the potential inaccuracy sources outlined in the preceding section apply to the DFTs. While most DFT design choices explicitly trade accuracy, they are hardly ever evaluated on accuracy. In Evaluation/Accuracy, we mark as *No* any paper that lacks a DFT-as-infrastructure kind of evaluation.

Generally speaking, while the potential for inaccuracy exists for each DFT, evaluations do not include accuracy study. Instead, authors tend to integrate the DFT with a particular use case (e.g., a mobile leakage alarm app in TaintDroid [Enck *et al.*, 2010], a logical data object system in Pebbles [Spahn *et al.*, 2014], and exploit detection in libdft and Argos [Portokalidis *et al.*, 2006]) and often evaluate some accuracy aspect of that particular use case. For example, the Pebbles paper evaluates the accuracy of the object system they develop atop TaintDroid; Argos evaluate effectiveness at detecting exploits. Only a few cases – T-BayesDroid [Tripp and Rubin, 2014] and H-BayesDroid [Tripp and Rubin, 2014] – include what we would call "generic" accuracy evaluation, which studies accuracy of a DFT system. Even in those cases, we find that evaluation is constrained to very simple microbenchmarks (e.g., DroidBench [Arzt *et al.*, 2014]), which do not capture the complexities of real applications.

In stark contrast with the limited accuracy evaluations, performance evaluations for these systems are often much more rigorous in the papers we surveyed. They generally include system overhead analyses in the context of both microbenchmarks and real applications; in some cases, they even perform detailed studies of the impact of varied design choices to performance.

For concreteness, consider the following DFTs:

- **TaintDroid** [Enck *et al.*, 2010] is an Android DFT that operates at Java interpreter level. It applies taints at coarse granularity: one taint per array, one taint per file, and one taint per interprocess communication (IPC) message. The system is evaluated thoroughly on, and acclaimed for,

its performance, but no accuracy study is presented (even for their specific use case). The system is available open-source and maintained in sync with Android releases. While the system was initially developed for one use case – multiple systems have leveraged it in recent years to implement new use cases [Spahn *et al.*, 2014; Tang *et al.*, 2012; Tripp and Rubin, 2014; Cox *et al.*, 2014].

• **TaintDroid-IPC** [Enck *et al.*, 2014], a version of TaintDroid that changes the granularity of one type of taints: IPC messages are tainted at byte level. The design is evaluated on performance but no accuracy study is presented. The system is released as part of the latest TaintDroid release and can be enabled with a compile-time flag.

• **TaintDroid-Array (Pebbles TaintDroid)** [Spahn *et al.*, 2014], a version of TaintDroid developed to support Pebbles, a system-level service in Android that recognizes the structure of application-level data objects – such as emails, documents, and bank accounts – by connecting their data items on disk based on information flows. TaintDroid-Array changes the granularity of array taints; each array element has a taint. It preserves coarse granularity for files and IPC. The design still permits inaccuracies yet it is only evaluated at the level of their specific use case.

These examples illustrate the lack of understanding accuracy in the DFT literature. This state is regrettable but understandable. It is regrettable because it leads to (1) poor understanding of the implications of various design decisions in a DFT system and (2) the potential for developing misguided designs. It is understandable because understanding accuracy is a difficult task, particularly in the context of real applications. We designed TaintMark to address these challenges and help DFT developers gain increased visibility into their designs' implications.

7.2.3 TaintMark Design Goals

Addressing aforementioned accuracy concerns, we formalize TaintMark's design goals and describe their implications:

1. *Support Real Applications:* TaintMark must be able to detect DFT accuracy bugs triggered by real, complex applications. However, it need *not* guarantee detection of all bugs or soundness of detected bugs.

2. *Support Debugging:* TaintMark must provide support for debugging the accuracy bugs it detects.

3. *Be DFT Agnostic:* Our methodology, if not the implementation, must be applicable to (almost) arbitrary DFT systems with limited effort.

Goal 1 reflects our conviction that DFT studies will require experimentation with real applications. This goal, which pervades our design, creates significant challenges, such as the impossibility to determine ground truth for information flows, limited testing coverage, and significant non-determinism. We overcome these challenges by opting for an imperfect but very practical tool; TaintMark may not detect all possible bugs, or guarantee to produce only correct bugs. This limitation is common to most bug finding tools. Despite limitations, our experience shows that TaintMark can yield many interesting results under multiple use cases. Moreover, TaintMark's support for debugging (Goal 2) provides an extra level of support for programmers wishing to assess exactly how accurate their own DFTs are.

Finally, Goal 3 illustrates our desire to build a general framework for understanding DFT accuracy. This goal, too, comes with challenging constraints: because we wish to minimize our dependency on the underlying DFT and operating system, we avoid making invasive changes to those systems, which in many cases could have greatly simplified our design. We next describe the TaintMark design, which stays true to these goals.

7.3 TaintMark Design and Implementation

To enable accuracy evaluations of DFTs on real applications, where true information flows are unknown, TaintMark tries to deduce flows using a DFT-independent blackbox differential testing. It feeds the application with differentiated inputs (e.g., different location values or contact lists) to test for impact on the application's outputs (such as files or network traffic). If the DFT's own tainting differs from TaintMark's information flow prediction for a particular input and output, TaintMark reports the case as an *accuracy bug* of the DFT.

This section describes our TaintMark design. For concreteness, we draw examples from our Android-based implementation of TaintMark, as well as its three instantiations on TaintDroid, TaintDroid-IPC, and TaintDroid-FG. The first two are all published [Enck *et al.*, 2010; Enck *et al.*, 2014] and publicly available; TaintDroid-FG is our own version of TaintDroid that employs fine-grained array and IPC taint tracking.

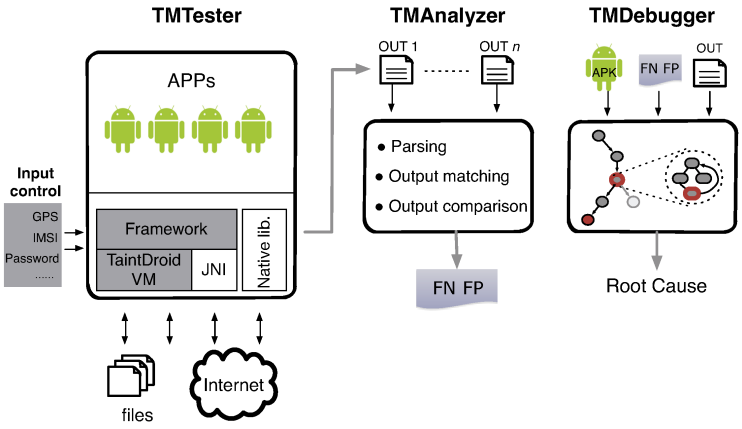


Fig. 7.1: The TaintMark Architecture.

7.3.1 TaintMark Architectural Overview

TaintMark consists of three major stages: (a) *measurement*, (b) *analysis*, and (c) *debugging* (optional). Figure 7.1 shows the components supporting each stage.

In the measurement stage, the developer interacts with the *TMTester* component to exercise real applications under differentiated inputs. TMTester produces logs for each input value consisting of the values and taint tags of the resulting outputs, as well as other information needed by subsequent stages. In the analysis stage, the developer uses the *TMAnalyzer* component, which takes in logs produced by TMTester and makes a determination of whether any false positives (FP) or false negatives (FN) have been triggered by the measurement stage. Any such cases are reported back to the developer as *accuracy bugs*. In the final debugging stage, the developer interacts with the *TMDebugger* component, which takes in logs produced by TMTester and provides the developer with support for tracing back the bug's origin.

DFT accuracy bug detection process, along with its mechanisms to deal with challenges raised by testing with real applications, is a core contribution of TaintMark. After describing our blackbox differential testing model, upon which bug detection relies, we detail the mechanisms that implement it in our practical DFT evaluation tool.

7.3.2 Blackbox Differential Testing Model

Our blackbox differential testing model is to provide different values to the application via the input source and observe whether these values result in changes of values at the output sinks. If we can observe (at least) two input values that lead to different output values, then we tentatively conclude that there is a data flow connecting the input and output. If we discover a flow, we expect to observe the input's taint among the output's taint, as given by the DFT system. In addition, if we can observe (at least) two input values that lead to the same output value, then we tentatively conclude that there is no information flow connecting the input and ouput. In that case, we expect the output's taint is empty even though the input is tainted.

We describe the above model more formally. We have a sensitive input location I (e.g., GPS location and password) and we can choose two different values (v_0, v_1) to the input I. For an output location O (e.g., network sends and file writes) and values $O(v_0)$, $O(v_1)$ that we observed from the output locations. $\tau()$ defines a function that returns taint from input or output locations provided by the DFT system. Suppose we can exactly match the same output location O across different execution runs for comparison and filter out the effects on output values that come from sources other than the input. Then, the testing procedure is defined as:

- For $v_0 \neq v_1$ and $O(v_0) \neq O(v_1)$, if $\tau(I) = \tau(O) \neq 0$ then we say it is a *correct* flow. Otherwise, we say it is a *false negative (FN)* flow.
- For $v_0 \neq v_1$ and $O(v_0) = O(v_1)$, if $\tau(I) = \tau(O) \neq 0$ then we say it is a *false positive (FP)* flow. Otherwise, we say it is a *correct* flow.

Our key technical challenge is to perform differential testing with real applications because the procedure is difficult to perform in practice due to non-determinism that violates the assumptions. The output value that we capture and examine can be influenced by different system states independent to the sensitive input that we specified for the experiment. As an instance, we can capture an output payload formatted as an HTTP GET request transmitted via the output location of *send()*. While it can have an entry (or field) that directly responds to values we provided via sensitive input (say GPS coordinates), it can also contain other entries (i.e., timestamp or sequence id) that vary and are not related to values provided via the sensitive input. This would make output matching across execution runs difficult since the output payload changes due to such non-deterministic inputs. We

handle non-determinism by examining multiple execution runs for the same input value. We discuss about this issue in Section 7.3.4.

7.3.3 TMTester

TMTester runs the application under differentiated inputs to generate the logs necessary for the next component of TMAnalyzer to detect accuracy bugs in a DFT by running multiple trials of each application on which the user wishes to test the DFT. For each trial, TMTester flashes the device to reset the system state. TMTester stubs the input sources of interest and returns a trial-specific value every time the application requests that input. In our Android-based prototype, TMTester stubs the GPS device, IMSI, and IMEI input sources, which were part of TaintDroid's initial input sources, plus any password fields. To stub these sources, we modified the Android framework to direct all calls to the input source to a long-running TMTester service, which supplies a value configured specifically for the current trial, tainted accordingly.

To test a DFT with TaintMark, on real applications, a developer creates automated scripts that exercise those applications. For example, to test TaintDroid, we wrote Robotium [Zadgaonkar, 2013] scripts for mobile applications to run. With the DFT enabled on the system, the developer launches TMTester, which will run each Robotium script several times, each time configuring the TMTester service to yield a particular set of values for the input sources. It runs each application a number of times (trials) for each input value, and it varies at most the value of one input source each time. For each trial, TMTester generates an `OUT` log, consisting of all outputs generated by that trial. These logs, whose formats are shown below, are used by TMAnalyzer to deduce the existence of a flow from the input to the output.

```
OUT log example:
  tid0:write:config.xml:tag0:<payload>
  tid1:send:www.google.com:tag1:<payload>
```

The `OUT` log captures all output activities intercepted by TMTester. In our Android-based prototype, we log activities on files and network connections (e.g., `read()`, `write()`, `send()`, `receive()`, which were part of TaintDroid's initial output sinks), plus the popular Webkit native library. For each output activity, TMTester logs all the information needed for TMAnalyzer to

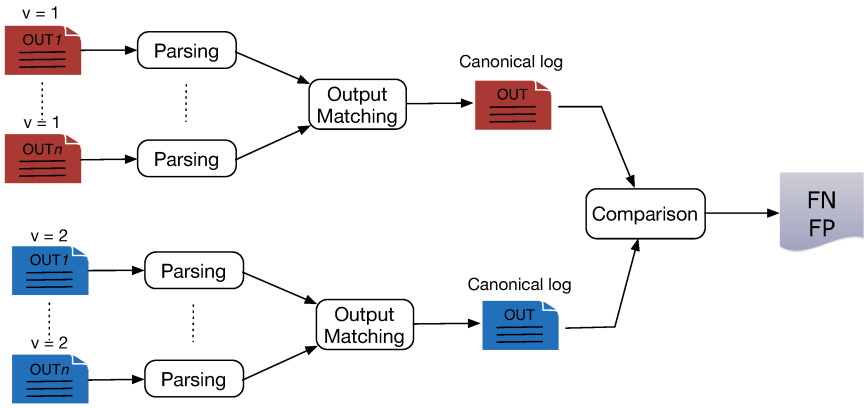


Fig. 7.2: TMAnalyzer consists of three main stages: parsing, output matching, and comparison of canonical logs for FP/FN decision.

match and compare outputs, including: the thread id, the operation, the output's location, the DFT's taint tag, and the operation's payload.

7.3.4 TMAnalyzer

Fig.7.2 shows the internals of TMAnalyzer. It takes as input a set of application execution logs from trials with both identical and different input values (the figure shows four logs, two for each input value). Processing these logs consists of three stages. First, TMAnalyzer applies a *Parsing* function that groups lines from a log into logical objects (e.g., HTTP requests and file writes). Second, TMAnalyzer produces a set of *canonical logs*, one per input value, by matching common outputs across execution runs. To identify the effect of non-determinism from the logs, such as background process outputs or fields within the object that consistently vary, it uses redundant trials available for each input value. Outputs that are not persistent in every execution are eliminated while varying fields are masked to facilitate the production of accurate canonical logs. Third, TMAnalyzer compares the canonical logs of different input values, and it matches their common outputs and compares the outputs' payloads to determine if the output is sensitive to the input. It compares its prediction with the DFT's own taints on the outputs to detect any accuracy bugs (FP or FN). We describe each stage in turn.

Parsing. Initially lines from an activity (e.g., HTTP requests, file writes) are ungrouped because the outputs from multiple activities are interleaved. We identify such activities (which we call logical

objects) with the following assumptions, which we have verified empirically: (a) a single thread does not output interleaving lines to the same destination (e.g., IP address, file name), (b) an XML output is always complete, and (b) a cache (e.g., Volley framework) always writes one object to a unique file (i.e., there is no filename re-use).

To facilitate producing canonical logs in output matching, we also exploit the standard structure that exists in network communication and file accesses. They follow well-defined protocols and payload formats. For example, most network traffic is over HTTP (or HTTPS); many files have well-defined formats, such as XML, JPEG, or JSON. This means that most output payloads have standard structure, which we can decode into individual fields (key-value pairs). Based on this observation, TMAnalyzer implements a number of decoders for common data structures (mostly using existing libraries).

For HTTP outputs, TMAnalyzer combines a sequence of lines to create a logical object based on the HTTP protocol that converts the fields of the request header to key-value pairs along with its corresponding body which is typically sent in different outputs. Before parsing files, TMAnalyzer aggregates the file content from the logs using the assumptions we mentioned before. For parsing files written to caches, we deal with two categories. For Volley, one image is stored as a file with a header prepended to the image. We use a decoder that ignores the header and uses the hash of the image for field comparison. We examined the source code of other caching libraries such as Android Universal Image Loader, Picasso and Web Image View and identified that they decode every image to a bitmap and save it to a file. As a result, TMAnalyzer only needs to use the hash of the file for the object comparison. For XML outputs, TMAnalyzer employs the ElementTree module, while for JSON outputs it uses the publicly available json parser module.

Output Matching. Our approach is to use logs from redundant trials under the same input value to cleanse logs of any non-deterministic activities before we compare output payloads under different input values. The hypothesis is that by running sufficient redundant trials for each input value, we can weed out most non-determinism, leaving only the activities related to the input. In our experiments, we have found that 10 redundant trials are sufficient to weed out asynchronous activities for most applications.

We compare all objects of the first execution with all objects of another execution and match objects based on their type and payload, forcing all matches to share the same payload structure. The

core idea of our approach is that matching objects must have the same structure (keys) and as few differences in the fields (values) as possible. As a result, we first match all identical objects using their hash value. For the rest we use the similarity of their payloads as a comparison metric and match objects that have the minimum number of differences in the fields as possible. To compare payloads of objects, we must address new non-determinism challenges at the payload level. Data written to a file or socket may include a counter, timestamp, or other constantly varying component, which will make it look like the object varies all the time. To address this issue, every time we match non-identical objects, we also mask their varying fields to make sure that they are not used in subsequent comparisons. This procedure is performed for the rest of the execution runs and results in the list of canonical objects. Section 7.4 shows that with just a few decoders, TMAnalyzer is able to recognize, match, and accurately compare output payloads in most applications we have tested.

Comparison of Canonical Objects. The last step of our approach is the comparison of the canonical objects to identify instances of false positives or false negatives. To do so we must first match canonical objects produced by different input values using the aforementioned matching methodology and then compare their payload. To conclude that we found an inaccuracy bug, we use the following criteria:

- If all the non-varying fields of tainted objects are the same, report this object pair as false positive
- If any of the non-varying fields of non-tainted objects are different, report this object pair as false negative
- For every canonical non-tainted object of one value that there is no corresponding canonical object in the list of the other value (i.e., no matching objects found), report it as *unmatched* false negative

7.3.5 Reverse Engineering with TMDebugger

By applying TaintMark on a DFT system, a user obtains a list of inaccuracy bugs that can be examined to gain insights regarding the source of the incorrect tainting. However, in some cases the inaccuracy reports are not sufficient to draw conclusions and a more thorough debugging methodology is needed. To this end, we provide a module (TMDebugger) which can assist the debugging

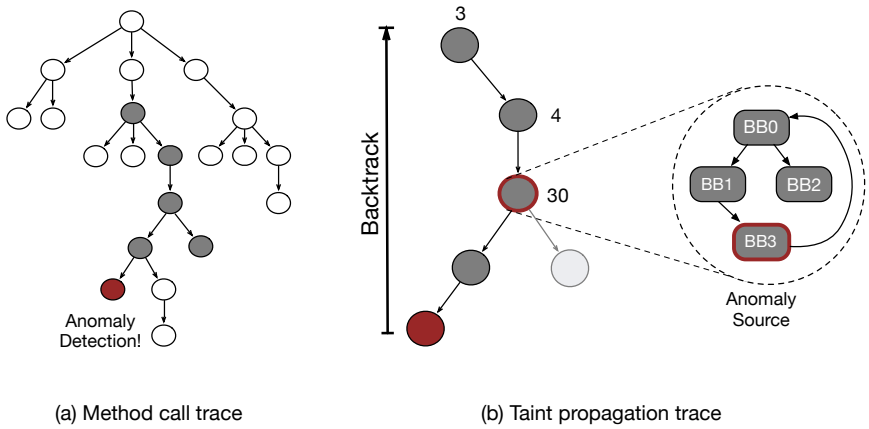


Fig. 7.3: TMDebugger Architecture

process and help the user identify the inaccuracy source for a given anomaly. TMDebugger consists of two parts, an extended version of TMTester that records the operations that are related to taint propagation and a utility that restores the flow of the taint for the anomaly.

Recording taint propagation: TMDebugger records how the taint propagates by maintaining information about method invocations and *taint* operations, for the target application process. For the former it records the instructions related to method invocation and return by instrumenting the Dalvik interpreter, while for the latter it records every instruction that transfer (non-zero) taint among its operands.

Restoring the taint flow: Using the extended log we describe above, TMDebugger is able to assist the debugging process by restoring the flow of the taint for an anomaly. To do this, it first creates the complete method call trace and converts it to tree. It then extracts the subtree that leads to the anomaly, making sure that all the methods of this subtree contain taint operations, thus revealing the flow of the taint. Fig. 7.3 shows the workflow of TMDebugger that restores the method call trace and extract the sub-tree connecting nodes that involve taint operation (gray nodes). For in-depth analysis, a method can be expanded to show basic block level propagation. The numbers annotations in Fig. 7.3 (b) represent the number of tainted operations in this method invocation.

Discovering the bug: Once the flow of the taint is restored, the user can use it to identify the cause of the anomaly. She examines the subtree starting from the anomaly and moving backwards, thus tracing back to the origin of the taint. Typically, the user looks for indications of suspicious taint

Table 7.2: Application Operations and Analysis Performance.

Apps	Source	Operations	Outputs	Time
Avantar	GPS	Search for food, move in the map	4554	35.656
BBC	Password	Scroll the headlines, click on article	244229	50.867
DrinkAdvisor	GPS	Log in, search for bars, move in the map	182409	211.03
Easytaxi	GPS	Log in, move in the map	3453	4.479
Fandango	GPS	Search for new movies from nearby theaters	2098	9.302
Grubhub	Password	Log in, search for restaurants, move in the map	15486	17.005
Imdb	Password	Log in, scroll new and all-time best movies	11710	36.35
Opentable	GPS	Search for restaurant, scroll result list	113348	50.203
Spotify	Password	Log in, scroll list of genres and songs	40729	31.78
Yelp	GPS	Log in, search for restaurant, move in map	97136	67.51

propagation within the examined methods (e.g., large number of taint operations) or across them (e.g., unexpected tainting). To facilitate a more fine-grained analysis, the taint flow within a method can be presented at the basic block level.

7.4 TaintMark Evaluation

Before demonstrating our tool's usefulness in the context of our use cases (section 7.5.1 and section 7.5.2), we evaluate the TaintMark tool itself, as well as the effectiveness of its core mechanisms. We seek to answer several questions:

Q1 *How efficient is TaintMark's analysis?*

Q2 *How many outputs can TaintMark digest?* and

Q3 *How accurate is TaintMark against ground truth?*

Q4 *How accurate is TaintMark against real applications?*

Workload and Evaluation Methodology. To answer the preceding questions, we run TaintMark on both microbenchmarks available in the literature [Arzt *et al.*, 2014] (where ground truth for information flows is known) and on ten real applications (where we try to validate its conclusions

manually). For each application, we wrote a Robotium [Zadgaonkar, 2013] script to exercise that application's functionality and run it using two different input values with ten redundant executions for each value. In each run, the test device was flashed to refresh the experiment environment in order to prevent the previoius execution's system state from affecting the current experiment. Table 7.2 presents the list of the applications we examined and a brief summary of the operations that our scripts performed on them. It contains the exercise operations, size of captured logs from TMTester in outputs and the TMAnalyzer analysis time in seconds. We ran TaintMark with various TaintDroid versions as the evaluated systems; unless noted otherwise, we ran it on stock TaintDroid 4.1.1. Our infrastructure consisted of: (1) 2 Google Nexus 7 tablets, which ran TMTester, and (2) one Debian 7 host (2.40 GHz six core Intel Xeon E5645 processor, 48GB of RAM), which ran TMAnalyzer.

TMAnalyzer Performance (Q1). We discuss the performance of TaintMark analysis for each application. Table 7.2 shows the number of outputs recorded by TMTester after exercising the applications and the time needed by TMAnalyzer to parse outputs into objects, match the objects among redundant executions to produce the canonical objects for each input value and finally compare the canonical objects to produce accuracy bug reports. The average running time for all the applications is close to 51 seconds which we consider acceptable. We empirically observe that the most heavyweight part of the analysis is parsing outputs into objects (mostly in the case of aggregated file outputs such as caches), which we believe can be improved by optimizing the process of aggregating outputs before decoding them into objects.

Digestible Outputs (Q2). TaintMark makes its decisions on object level to make sure that its inaccuracy reports are not affected by fragmentation in the output level, such as POST requests that send the header and the body in separate outputs. In this paragraph, we evaluate the ability of TMAnalyzer to parse outputs into objects. Table 7.3 shows how well TaintMark digests the outputs produced by the applications we exercised. TaintMark is able to digest 77% of the total outputs however if we exclude two applications (BBC and Grubhub) that TaintMark performs badly, the percentage raises to 92%. BBC uses an old version of the Volley framework that used a different caching scheme and therefore our Volley decoder failed to parse the outputs into objects. We believe that a decoder for this version of Volley can be easily implemented and would remove the vast majority of unparsed outputs because Volley writes most of its header one byte at a time. Almost

Table 7.3: Digestible Outputs. The Caches category contains the Volley decoder and the generic cache decoder. The Other category contains the XML and JSON decoders.

Apps	Outputs		Objects		
	Total	Unparsed	HTTP	Caches	Other
Avantar	4554	320	1200	0	896
BBC	244229	242671	388	0	250
DrinkAdvisor	182409	37764	3054	829	546
EasyTaxi	3453	99	1399	0	214
Fandango	2098	0	380	0	438
Grubhub	15486	9314	1791	0	485
Imdb	11710	220	2723	1521	393
Opentable	113348	5972	1638	465	562
Spotify	40729	4516	1983	3002	267
Yelp	97136	13795	2537	892	700

all of the unparsed outputs of Grubhub are file writes that come from a single library (Crashlytics), for which TaintMark does not have a decoder yet.

In general, adding a decoder is a simple operation involving only a few development hours in most cases. Three out of five parsers we used (HTTP, XML and JSON) were open-source third party libraries. For the other two parsers we built to support these applications it took less than 3 hours, since we only had to identify their structure and slightly modify the original code to parse the object. Thus, we deem our approach of parsing outputs as amenable and effective. For the rest of the evaluation section, we will focus solely on object level granularity and the objects that TaintMark was able to digest.

TaintMark Accuracy against ground truth (Q3). We evaluate TaintMark's own accuracy using both microbenchmarks and real applications. For the microbenchmark evaluation, we used DroidBench [Arzt *et al.*, 2014], a collection of very simple applications – similar to unit tests – that test specific information flow patterns, which DFT systems should be able to detect. For all 70 applications that we tested, TaintMark was able to correctly detect the relation between the input and the output and classify TaintDroid's own taints as true positives (TP), true negatives (TN), false positives (FP) and false negatives (FN). These applications test a number of different information flow patterns such implicit flows, no leak, direct leak and other. Overall, we can report that the very

simple applications included in DroidBench pose no issues to TaintMark, because they lack any level of non-determinism and noise that is prevalent in real applications.

TaintMark Accuracy against real applications (Q4). To understand TaintMark's accuracy in more realistic cases, we also ran TaintMark with a set of real applications, for which we took the time to manually inspect all of its reports and cluster them into categories. Unfortunately, for these cases it is very difficult to identify the ground truth in order to decide whether an inaccuracy report is actually correct or incorrect, because of the prevelance of fields that are not in a plaintext format (such encoded strings or binaries) or that convey indirect information about the input (e.g., the postal code of the location of the user).

To categorize each report, we examined the payload of the corresponding canonical objects along with the original payloads that were involved in the matching process. For false negatives we used grep command to identify cases where the input value was transmitted as plaintext. Whenever this failed, we manually examined all the fields of the payloads, paying special attention to the fields that influenced the decision of TMAnalyzer to identify whether they are related to the input value. If the fields were not related to the input value, then we categorize the report as incorrect. Otherwise we categorize it as correct. For false positives we examined all the masked fields in the canonical object and looked for fields that contain information related to the input value. If there were none, then we consider the report correct, otherwise we consider it incorrect. In both cases, whevener we encounter a field that affects TaintMark decision and is in a format that we cannot understand (such as encoded, binary etc), we categorize it as *unconfirmed*.

Table 7.4 shows the reports produced by TMAnalyzer and our categorization of them. From the table, evaluations for FP and FN contain sub-column entries of **C**orrect, **I**ncorrect, **U**nconfirmed. **UM** is only for FN result to represent unmatched but tainted outputs.

We discuss some examples of correct, incorrect and unconfirmed cases that we encountered during the clustering process. The most clear example of correct false positive is when the objects are matched identically to each other, something that happened in many reports of Spotify. In this case, we can be certain that the report is correct because the objects were not affected by the input value. Yelp had many incorrect false negative reports because during matching GET requests for images, the filename path was masked (identified as a field that varies) because it was encoded. This caused incorrect comparisons between different image requests that should not be matched. Imdb

exhibited something similar but in this case the masked filename path contained an encoded string, which led us to categorize it as unconfirmed false positive.

Table 7.4: Tool Evaluation; Manual verification result regarding TMAnalyzer's accuracy evaluation.

Apps	FP			FN			
	C	I	U	C	UM	I	U
Avantar	0	3	1	11	8	1	0
BBC	0	0	0	0	0	0	1
DrinkAdvisor	0	0	0	0	42	0	1
EasyTaxi	0	0	0	0	4	0	0
Fandango	0	0	0	0	0	0	0
Grubhub	8	0	9	0	1	0	4
Imdb	0	0	22	0	3	0	3
OpenTable	0	11	3	0	1	0	0
Spotify	101	1	0	0	1	0	1
Yelp	0	9	2	0	32	10	0

7.5 Accuracy Evaluation for DFT systems

To see how the different design choices have influence on the accuracy results, we extended the previous experiment to include TaintDroid based systems of finer granularity (TaintDroid-IPC, TaintDroid-FG). With these system TMAnalyzer first evaluates and compares the accuracy of three different implementations. We then move onto case studies that further analyze the accuracy bugs and summary the major reasons behind those.

7.5.1 TaintDroid Granularity on Propagation Accuracy

We compare the different TaintDroid implementation to gain insight on how each one's design choice impacts the accuracy of the DFT system. Referring to the report produced by TaintMark and using it as an indication, we start debugging process which can lead to useful conclusion about the error source and related inaccurate tainting pattern.

Table 7.5: Inaccuracy Reports for TaintDroid implemetations.

Apps	TaintDroid			TaintDroid-IPC			TaintDroid-FG		
	FP	**FN**	**INC**	**FP**	**FN**	**INC**	**FP**	**FN**	**INC**
Avantar	4	20	0	0	23	0	0	27	0
BBC	0	1	0	0	2	0	0	1	0
DrinkAdvisor	0	43	1	0	40	1	0	13	0
EasyTaxi	0	4	0	0	2	0	0	2	0
Fandango	0	0	4	0	2	0	0	5	0
Grubhub	17	5	0	20	4	0	0	46	0
Imdb	22	6	38	30	4	0	0	30	0
OpenTable	14	1	31	3	2	41	1	6	0
Spotify	102	2	2	87	9	0	0	64	0
Yelp	11	42	6	6	47	0	7	50	0

Table 7.5 shows the accuracy evaluation result produced by TaintMark for different TaintDroid implementations. From the result, TaintMark is able to report instances of false positives, false negatives and ases of inconsistent tainting. For all TMAnalysis reports, TaintMark users can further investigate their root causes by using TMDebugger.

One interesting observation that we made is that in a number of applications, objects showed *inconsistent* taint behavior across the multiple executions for the same input value. In this case, we consider that an object is *inconsistent* if it gets taint in some executions but in others does not. For the entries, we only counted the object matches to the hash level in order to strictly exclude the cases where object matching errors can get involved. In section 7.5.2 we further discuss about inconsistent tainting with an application (Opentable).

Table 7.5 can also be used to draw to gain insights on the effect of each optimization. For instance, there is a large number of false positive reports in Spotify for both TaintDroid and TaintDroid-IPC, which however is eliminated in TaintDroid-FG, something that indicates that the cause of the inaccuracy is related to coarse-grained array tainting. Similar conclusions can be reached for Imdb and Grubhub.

7.5.2 Investigating Inaccuracy Bugs

Using the TMAnalyzer reports in section 7.5.1 as an investigation starting point, we used TMDebugger to find the root cause of a number of reports. In this section we describe some for our findings and explain which design choice is responsible for the observed inaccuracy.

Opentable allows users to search for restaurants with vacancies close to the user location. It uses the Volley framework for transparent caching of restaurant images. The reports of TMAnalyzer revealed a large number of inconsistent tainting and one false positive report for a Volley cache file that we decided to investigate further.

Applying the TMDebugger methodology on Opentable, revealed multiple sources of over-tainting contributing to these anomalies. First, the coarse-grained taint tracking implementation of the Binder, an Android interprocess communication mechanism, leads to incorrect object reference tainting and this in turn leads to file level over-tainting. Moreover, TMDebugger showed that the problem persisted in TaintDroid-IPC which led to the discovery of a bug in the byte-level taint tracking implementation of the Binder. We first discuss our findings regarding TaintDroid and then describe the implementation bug we discovered in TaintDroid-IPC.

The first occurrence of over-tainting is met when an object that holds the available restaurants (along with the GPS location of the user and the request time) is transferred from one Activity to another through the Binder, an Android interprocess communication mechanism. TaintDroid tracks Binder transfers in a coarse-grained manner by aggregating the taint of the members and propagating one taint per message. This leads to over-tainting because every member of the object becomes tainted (including ones that were originally clean of taint).

Volley employs a searchAdapter object to request images from the list of restaurants based on their availability time using the (now tainted) request time for sorting. It then conditionally updates one of its member variables if the earliest availability is earlier than its current value. This subsequently leads erroneously tainting the searchAdapter object reference and as a consequence all of its members. Consequently, When the restaurant images are stored for caching, searchAdapter prepends them with one of its member variables that acts as a cache file key which explains the false positive report we obtained by TMAnalyzer.

Given that the restaurant availability times can change independently of our executions, searchAdapter can become contaminated at different points within the execution flow, thus explaining the incon-

sistent tainting reports.

Finally, even though TMAnalyzer did not report instances of over-tainting due to files because we reverted to a clean state before every execution, the coarse granularity of file tainting would force TaintDroid to consider the whole file as tainted, further increasing the effect over-tainting.

TMDebugger pinpointed Binder transfers as the initial source of over-tainting, thus one might expect that the problem would not exist in the extensions of TaintDroid that implement byte-level Binder tainting. However, TMAnalyzer revealed that the problem still existed. To clarify the situation, we again applied the TMDebugger methodology to TaintDroid-IPC and discovered a bug in the implementation of the byte-level Binder tainting that incorrectly calculates the offset of the internal array that holds the tags. The bug was reported to the TaintDroid developers and was subsequently patched.

BBC Mobile News is a news application that allows the user to explore the news headlines and read their corresponding articles. TMAnalyzer produced no false positive report for this application in any of the TaintDroid based systems, because the application produced no tainted output when we restored the device to a clean state before every execution. Such a taint-free application enabled us to test how the system state affects the number of tainted outputs produced by TaintDroid, an aspect often overseen when discussing the accuracy of DFT system. To examine the effect of taint residues, we ran the same Robotium script as in the original experiments but on a device that was previously used by multiple applications. We observed that the number of tainted outputs increased significantly, reaching to the order of tens of thousands. We decided to investigate these outputs and search for their taint source using TMDebugger.

TMDebugger revealed that the source of the over-tainting problem was *SharedPreferences*, an object that maintains information about the global state of the system, such as the orientation of the screen, the keyboard visibility etc This information along with application-specific settings is stored in an XML file that is manipulated multiple times in every execution to read and update the current settings. Since TaintDroid keeps one taint value per file, this XML file is contaminated and therefore every file read leads to over-tainting.

A similar case is **Fandango**, an application that presents information about movies and theaters, for which TMAnalyzer also produced no false positive reports. Encouraged by our BBC finding, we performed the same type of experiment on this application and again observed a significant rise

in the number of tainted outputs. We applied the TMDebugger methodology to some of its outputs to identify sources of over-tainting. TMDebugger pinpointed *Adobe AppMeasurement (ADMS)*, a library that gathers statistics related to the application activity (e.g., usage), as a source of over-tainting. Initially the taint originates from a global object of ADMS that holds statistics for the previous execution. These statistics are written to an XML file which subsequently becomes a major source of over-tainting, similar to the BBC case. Finally, we were able to identify similar over-tainting patterns in other analytics libraries such as Google Analytics.

DrinkAdvisor searches for bars close to the user location and imports the *Google Maps* API to graphically display the search results. We exercise this API by dragging the center of the map to different locations to render new results. Every time the center of the map is moved, a POST request message is sent in two separate output, one holding the header and another holding the body. The header contains no locational information, while the body holds the new map coordinates (tainted with the user GPS taint) in a binary format. In the case of TaintDroid and TaintDroid-IPC, the DFT systems produced two leakage reports, while TaintDroid-FG produced only one since it accurately tracks the propagation of the taint and identifies that only the body of the request should be tainted. In this case, the source of over-tainting is the coarse-granularity of array taint tracking. More importantly, this behavior is common in every application that imports the Google Maps API.

Part VI

Conclusion

Chapter 8

Conclusions

In this dissertation we explored *efficiency* and *accuracy* aspects of the DFT techology.

As a first step, we presented our foundation tool libdft, a practical dynamic DFT platform that is at once *fast*, *reusable*, and applicable to *commodity software and hardware*. We also investigated the reasons that DFT tools based on DBI frameworks frequently perform badly, and presented practices that should be avoided by authors of such tools.

TFA is a novel methodology that combines dynamic and static analysis to improve the performance overhead of DFT. Our approach separates data tracking logic from program logic, and represents it using a Taint Flow Algebra. Inspired by optimizing compilers, we apply various code optimization techniques, like dead code elimination and liveness analysis, to eliminate redundant tracking operations and minimize interference with the instrumented program. We incorporated our optimizations with our baseline DFT implementation – libdft and evaluated its performance on various applications. We then introduced ShadowReplica, a new and efficient approach that further accelerates DFT by running application and data tracking logics utilizing spare CPU cores. We again resort to a combination of off-line dynamic and static analysis of the application to minimize the data that need to be communicated for decoupling the analysis, and optimize the code used to perform it. Furthermore, we design and tune a shared ring buffer data structure for efficiently sharing data between threads on multi-core CPUs. While the two proposed optimization approaches could achieve significant performance improvement over libdft, they also preserve all tracking operations of the DFT framework, thus not sacrificing any accuracy guarantees. They do not require additional resources neither do they restrict or remove functionality. This is due to the off-line static

analysis which takes advantage of the rich theory on basic block optimization and data flow analysis, done in the context of compilers, to argue the safety and correctness of our algorithm using a formal framework.

TaintMark is a framework to evaluate the accuracy of DFT systems by implementing a methodology to infer ground truth regarding data tracking for real world applications. TaintMark uses differential testing with mechanisms to deal with challenges raised by testing with real applications to produce reports of potential inaccuracy bugs. These reports are then analyzed using TaintMark's reverse engineering methodology and tool to discover the root cause. Using this approach we were able to find a number of inaccuracy patterns. The design of TaintMark makes it possible to port it to new DFT systems with moderate effort and by releasing it as an open-source tool we hope to help DFT developers understand the impact of their design choices and help users dissect potential inaccuracy bugs.

We believe that our research strikes a balance between the performance and accuracy aspects of the well explored technology of DFT. Although the overall performance impact of DFT remains significant and our accuracy measurement methodology has not yet reached the point that we can claim it is complete, we hope that our research will bring the technology closer to becoming practical for certain environments.

Part VII

Bibliography

Bibliography

[Abadi *et al.*, 2005] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security*, CCS '05, pages 340–353, New York, NY, USA, 2005. ACM.

[Aho *et al.*, 2006] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, & Tools*. Addison Wesley, 2006.

[Arzt *et al.*, 2014] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *Proc. of PLDI*, 2014.

[Attariyan *et al.*, 2012] Mona Attariyan, MIchael Chow, and Jason Flinn. X-ray: Automating root-cause diagnosis of performance anomalies in production software. In *Proc. of OSDI*, 2012.

[Barham *et al.*, 2003] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. *SIGOPS Oper. Syst. Rev.*, 37(5):164–177, October 2003.

[Bellard, 2005] Fabrice Bellard. QEMU, a fast and portable dynamic translator. In *Proc. of the USENIX ATC*, pages 41–46, April 2005.

[Benameur *et al.*, 2013] Azzedine Benameur, Nathan S. Evans, and Matthew C. Elder. Minestrone: Testing the soup. In *Proceedings of the 6th Workshop on Cyber Security Experimentation and Test (CSET)*, 2013.

[Bosman *et al.*, 2011] Erik Bosman, Asia Slowinska, and Herbert Bos. Minemu: The World's Fastest Taint Tracker. In *Proc. of the* 14^{th} *RAID*, pages 1–20, 2011.

[Bruening and Zhao, 2011] D Bruening and Q Zhao. Practical Memory Checking with Dr. Memory. In *Proc. of the* 9^{th} *CGO*, pages 213–223, 2011.

[Bruening, 2004] Derek L. Bruening. *Efficient, Transparent, and Comprehensive Runtime Code Manipulation*. PhD thesis, Cambridge, MA, USA, 2004. AAI0807735.

[Chen and Chen, 2013] Yufei Chen and Haibo Chen. Scalable deterministic replay in a parallel full-system emulator. In *Proc. of PPoPP*, 2013.

[Chen *et al.*, 2008] S Chen, M Kozuch, T Strigkos, B Falsafi, P.B Gibbons, T.C Mowry, V Ramachandran, O Ruwase, M Ryan, and E Vlachos. Flexible Hardware Acceleration for Instruction-Grain Program Monitoring. In *Proc. of the* 35^{th} *ICSA*, pages 377–388, 2008.

[Chipounov *et al.*, 2011] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. S2E: a platform for in-vivo multi-path analysis of software systems. In *Proc. of ASPLOS*, 2011.

[Chow *et al.*, 2004] Jim Chow, Ben Pfaff, Tal Garfinkel, Kevin Christopher, and Mendel Rosenblum. Understanding Data Lifetime via Whole System Simulation. In *Proc. of the* 13^{th} *USENIX Security*, pages 321–336, 2004.

[Chow *et al.*, 2008] J Chow, T Garfinkel, and PM Chen. Decoupling dynamic program analysis from execution in virtual environments. In *Proc. of USENIX ATC*, 2008.

[Clark *et al.*, 2001] David Clark, Sebastian Hunt, and Pasquale Malacaria. Quantitative analysis of the leakage of confidential data. In *Proc. of QAPL*, 2001.

[Clark *et al.*, 2005] David Clark, Sebastian Hunt, and Pasquale Malacaria. Quantified interference for a while language. *Electron. Notes Theor. Comput. Sci.*, 2005.

[Clarkson *et al.*, 2005] Michael R. Clarkson, Andrew C. Myers, and Fred B. Schneider. Belief in information flow. In *Proc. of CSFW*, 2005.

[Clause *et al.*, 2007] James Clause, Wanchun Li, and Alessandro Orso. Dytan: A Generic Dynamic Taint Analysis Framework. In *Proc. of the 2007 ISSTA*, pages 196–206, 2007.

[Cordy, 2006] James R. Cordy. The txl source transformation language. *Sci. Comput. Program.*, 61(3):190–210, August 2006.

[Costa *et al.*, 2005] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham. Vigilante: End-to-end containment of internet worms. In *Proc. of SOSP*, 2005.

[Cox *et al.*, 2014] Landon P Cox, Peter Gilbert, Geoffrey Lawler, Valentin Pistol, Ali Razeen, Bi Wu, and Sai Cheemalapati. SpanDex: secure password tracking for android. In *Proc. of Usenix Security*, 2014.

[Crandall and Chong, 2004] Jedidiah R. Crandall and Frederic T. Chong. Minos: Control Data Attack Prevention Orthogonal to Memory Model. In *Proc. of the* 37^{th} *MICRO*, pages 221–232, 2004.

[Dalton *et al.*, 2007] Michael Dalton, Hari Kannan, and Christos Kozyrakis. Raksha: A flexible information flow architecture for software security. In *Proc. of ISCA*, 2007.

[Dalton *et al.*, 2008] Michael Dalton, Hari Kannan, and Christos Kozyrakis. Real-World Buffer Overflow Protection for Userspace & Kernelspace. In *Proc. of the* 17^{th} *USENIX Security*, pages 395–410, 2008.

[Dietz *et al.*, 2012] Will Dietz, Peng Li, John Regehr, and Vikram Adve. Understanding integer overflow in C/C++. In *Proceedings of the 34th International Conference on Software Engineering (ICSE)*, 2012.

[Enck *et al.*, 2010] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *Proc. of OSDI*, 2010.

[Enck *et al.*, 2014] William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Trans. Comput. Syst.*, 2014.

[Festa *et al.*, 1999] Paola Festa, Panos M Pardalos, and Mauricio GC Resende. Feedback set problems. *Handbook of Combinatorial Optimization*, 4:209–258, 1999.

[Ford and Cox, 2008] Bryan Ford and Russ Cox. Vx32: Lightweight User-level Sandboxing on the x86. In *Proc. of the 2008 USENIX ATC*, pages 293–306, 2008.

[Ha *et al.*, 2009] Jungwoo Ha, Matthew Arnold, Stephen M. Blackburn, and Kathryn S. McKinley. A concurrent dynamic analysis framework for multicore hardware. In *Proc. of OOPSLA*, 2009.

[Henderson *et al.*, 2014] Andrew Henderson, Aravind Prakash, Lok Kwong Yan, Xunchao Hu, Xujiewen Wang, Rundong Zhou, and Heng Yin. Make it work, make it right, make it fast: Building a platform-neutral whole-system dynamic binary analysis platform. In *Proc. of ISSTA*, 2014.

[Hex-Rays, cited Aug 2013] Hex-Rays. The IDA Pro Disassembler and Debugger, cited Aug. 2013. `http://www.hex-rays.com/products/ida/`.

[Ho *et al.*,] Alex Ho, Michael Fetterman, Christopher Clark, Andrew Warfield, and Steven Hand. Practical Taint-based Protection using Demand Emulation. In *Proc. of the 2006 EuroSys*, pages 29–41.

[Jung *et al.*, 2008] Jaeyeon Jung, Anmol Sheth, Ben Greenstein, David Wetherall, Gabriel Maganis, and Tadayoshi Kohno. Privacy oracle: A system for finding application leaks with black box differential testing. In *Proc. of CCS*, 2008.

[Kang *et al.*, 2011] Min Gyung Kang, Stephen McCamant, Pongsin Poosankam, and Dawn Song. DTA++: Dynamic Taint Analysis with Targeted Control-Flow Propagation. In *Proc. of the* 18^{th} *NDSS*, 2011.

[Kemerlis *et al.*, 2012] Vasileios P. Kemerlis, Georgios Portokalidis, Kangkook Jee, and Angelos D. Keromytis. Libdft: Practical dynamic data flow tracking for commodity systems. In *Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments*, VEE '12, pages 121–132, New York, NY, USA, 2012. ACM.

[Kim and Keromytis, 2009] Hyung C. Kim and Angelos Keromytis. On the Deployment of Dynamic Taint Analysis for Application Communities. *IEICE Transactions on Information and Systems*, E92-D(3):548–551, 2009.

[Lam and Chiueh, 2006] Lap Chung Lam and Tzi-cker Chiueh. A General Dynamic Information Flow Tracking Framework for Security Applications. In *Proc. of the* 22^{nd} *ACSAC*, pages 463–472, 2006.

[Lamport, 1983] Leslie Lamport. Specifying Concurrent Program Modules. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 1983.

[Lattner and Adve, 2004] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the international symposium on Code generation and optimization (CGO)*, 2004.

[Lattner *et al.*, 2007] Chris Lattner, Andrew Lenharth, and Vikram Adve. Making context-sensitive points-to analysis with heap cloning practical for the real world. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2007.

[Lecuyer *et al.*, 2014] Mathias Lecuyer, Guillaume Ducoffe, Francis Lan, Andrei Papancea, Theofilos Petsios, Riley Spahn, Augustin Chaintreau, and Roxana Geambasu. XRay: Enhancing the Web's Transparency with Differential Correlation. In *Proc. of Usenix Security*, 2014.

[Lee *et al.*, 2013] Kyu Hyung Lee, Xiangyu Zhang, and Dongyan Xu. High accuracy attack provenance via binary-based execution partition. In *Proc. of NDSS*, 2013.

[Luk *et al.*, 2005] CK Luk, R Cohn, R Muth, H Patil, A Klauser, G Lowney, S Wallace, VJ Reddi, and K Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proc. of PLDI*, 2005.

[McCamant and Ernst, 2008] Stephen McCamant and Michael D. Ernst. Quantitative information flow as network flow capacity. In *Proc. of PLDI*, 2008.

[Millen, 1987] Jonathan K. Millen. Covert channel capacity. In *Proc. of Oakland*, 1987.

[MIT,] CWE - Common Weakness Enumeration. `http://cwe.mitre.org/`.

[Molnar *et al.*, 2009] David Molnar, Xue Cong Li, and David A. Wagner. Dynamic test generation to find integer bugs in x86 binary linux programs. In *Proceedings of the 18th USENIX Security Symposium*, 2009.

[Moore, 2013] Scott Moore. thinkmoore/llvm-deps. `https://github.com/thinkmoore/llvm-deps`, 2013. (Visited on 06/07/2014).

[Necula *et al.*, 2002] George C. Necula, Scott McPeak, Shree Prakash Rahul, and Westley Weimer. Cil: Intermediate language and tools for analysis and transformation of c programs. In *Proceedings of the 11th International Conference on Compiler Construction*, CC '02, pages 213–228, London, UK, UK, 2002. Springer-Verlag.

[Nethercote and Seward, 2007] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proc. of PLDI*, 2007.

[Nethercote, 2004] Nicholas Nethercote. *Dynamic Binary Analysis and Instrumentation*. PhD thesis, Computer Laboratory, University of Cambridge, United Kingdom, November 2004.

[Newsome and Song, 2005] J Newsome and D Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proc. of NDSS*, 2005.

[Newsome *et al.*, 2009] James Newsome, Stephen McCamant, and Dawn Song. Measuring channel capacity to distinguish undue influence. In *Proc. of PLAS*, 2009.

[Nightingale *et al.*, 2008] Edmund B Nightingale, Daniel Peek, Peter M Chen, and Jason Flinn. Parallelizing security checks on commodity hardware. In *Proc. of ASPLOS*, 2008.

[Portokalidis and Bos, 2008] Georgios Portokalidis and Herbert Bos. Eudaemon: Involuntary and On-Demand Emulation Against Zero-Day Exploits. In *Proc. of the 2008 EuroSys*, pages 287–299, 2008.

[Portokalidis *et al.*, 2006] Georgios Portokalidis, Asia Slowinska, and Herbert Bos. Argos: an Emulator for Fingerprinting Zero-Day Attacks. In *Proc. of the 2006 EuroSys*, pages 15–27, 2006.

[Portokalidis *et al.*, 2010] Georgios Portokalidis, Philip Homburg, Kostas Anagnostakis, and Herbert Bos. Paranoid Android: Versatile protection for smartphones. In *Proc. of ACSAC*, 2010.

[Qin *et al.*, 2006] Feng Qin, Cheng Wang, Zhenmin Li, Ho-seop Kim, Yuanyuan Zhou, and Youfeng Wu. LIFT: A Low-Overhead Practical Information Flow Tracking System for De-

tecting Security Attacks. *MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, LIFT:2006, December 2006.

[Ruwase *et al.*, 2008] Olatunji Ruwase, Phillip B. Gibbons, Todd C. Mowry, Vijaya Ramachandran, Shimin Chen, Michael Kozuch, and Michael Ryan. Parallelizing Dynamic Information Flow Tracking. In *Proc. of the* 20^{th} *SPAA*, pages 35–45, 2008.

[Saxena *et al.*, 2008] Prateek Saxena, R Sekar, and Varun Puranik. Efficient Fine-Grained Binary Instrumentation with Applications to Taint-Tracking. In *Proc. of the* 6^{th} *CGO*, pages 74–83, 2008.

[Schwartz *et al.*, 2010] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask). In *Proc. of the* 31^{th} *IEEE S&P*, pages 317–331, 2010.

[Sen *et al.*, 2014] Shayak Sen, Saikat Guha, Anupam Datta, Sriram K Rajamani, Janice Tsai, and Jeannette M Wing. Bootstrapping Privacy Compliance in Big Data Systems. In *Proc. of Oakland*, 2014.

[Slowinska and Bos, 2009] A Slowinska and H Bos. Pointless tainting?: evaluating the practicality of pointer tainting. In *Proc. of EuroSys*, 2009.

[Slowinska *et al.*, 2011] Asia Slowinska, Traian Stancescu, and Herbert Bos. Howard: a dynamic excavator for reverse engineering data structures. In *Proc. of NDSS*, 2011.

[Spahn *et al.*, 2014] Riley Spahn, Jonathan Bell, Michael Lee, Sravan Bhamidipati, Roxana Geambasu, and Gail Kaiser. Pebbles: Fine-grained data management abstractions for modern operating systems. In *Proc. of OSDI*, 2014.

[Suh *et al.*, 2004] G. Edward Suh, Jae W. Lee, David Zhang, and Srinivas Devadas. Secure Program Execution via Dynamic Information Flow Tracking. In *Proc. of the* 11^{th} *ASPLOS*, pages 85–96, 2004.

[Tang *et al.*, 2012] Yang Tang, Phillip Ames, Sravan Bhamidipati, Ashish Bijlani, Roxana Geambasu, and Nikhil Sarda. Cleanos: Limiting mobile data exposure with idle eviction. In *Proc. of OSDI*, 2012.

[Tripp and Rubin, 2014] Omer Tripp and Julia Rubin. A bayesian approach to privacy enforcement in smartphones. In *Proc. of Usenix Security*, 2014.

[Venkataramani *et al.*, 2008] Guru Venkataramani, Ioannis Doudalis, Yan Solihin, and Milos Prvulovic. Flexitaint: A Programmable Accelerator for Dynamic Taint Propagation. In *Proc. of the* 14^{th} *HPCA*, pages 173–184, 2008.

[Wahbe *et al.*, 1993] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. In *Proc. of SOSP*, 1993.

[Wallace and Hazelwood, 2007] S Wallace and K Hazelwood. Superpin: Parallelizing dynamic instrumentation for real-time performance. In *Proc. of CGO*, 2007.

[Wang *et al.*, 2012] Xi Wang, Haogang Chen, Zhihao Jia, Nickolai Zeldovich, and M Frans Kaashoek. Improving integer security for systems with KINT. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 2012.

[Xu *et al.*, 2006] Wei Xu, Sandeep Bhatkar, and R. Sekar. Taint-Enhanced Policy Enforcement: A Practical Approach to Defeat a Wide Range of Attacks. In *Proc. of the* 15^{th} *USENIX Security*, pages 121–136, 2006.

[Yin *et al.*, 2007] Heng Yin, Dawn Song, Manuel Egele, Christopher Kruegel, and Engin Kirda. Panorama: capturing system-wide information flow for malware detection and analysis. In *Proc. of CCS*, 2007.

[Zadgaonkar, 2013] Hrushikesh Zadgaonkar. *Robotium Automated Testing for Android*. Packt Publishing, 2013.

[Zeldovich *et al.*, 2006] Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. Making Information Flow Explicit in HiStar. In *Proc. of the* 7^{th} *OSDI*, pages 263–278, 2006.

[Zhang *et al.*, 2010] Chao Zhang, Tielei Wang, Tao Wei, Yu Chen, and Wei Zou. Intpatch: Automatically fix integer-overflow-to-buffer-overflow vulnerability at compile-time. In *Proceedings of the 15th European Symposium on Research in Computer Security (ESORICS)*, 2010.

[Zhao *et al.*, 2008] Q Zhao, I Cutcutache, and WF Wong. PiPA: pipelined profiling and analysis on multi-core systems. In *Proc. of CGO*, 2008.

[Zhu *et al.*, 2011] David Zhu, Jaeyeon Jung, Dawn Song, Tadayoshi Kohno, and David Wetherall. TaintEraser: Protecting sensitive data leaks using application-level taint tracking. In *SIGOPS Oper. Syst. Rev.*, 2011.

Part VIII

Appendices

Appendix

Correctness Proofs for TFA Optimizations

.

1 Formal Definitions and Semantics

1.1 The Range Operator

$$\frac{\Sigma \vdash l = [0, size(\Sigma)]}{\Sigma \vdash r(constant) \Downarrow l}\ \textbf{Constant} \qquad \frac{\Sigma \vdash l = [def(reg_{N-1}), def(reg_{N+1})]}{\Sigma \vdash r(reg_N) \Downarrow l}\ \textbf{Register}$$

$$\frac{\Sigma \vdash l = \bigcap_{\forall reg \in mem_N} r(reg) \cap [def(mem_{N-1}), def(mem_{N+1})]}{\Sigma \vdash r(mem_N) \Downarrow l}\ \textbf{Memory} \qquad \frac{\Sigma \vdash l = \bigcap_{\forall var \in expr} r(var)}{\Sigma \vdash r(expr) \Downarrow l}\ \textbf{Expression}$$

$$\frac{\Sigma \vdash l = r(var) \cap r(expr) \quad \Sigma' = \Sigma[l \leftarrow var ::= expr]}{\Sigma \vdash r(var ::= expr) \rightsquigarrow \Sigma'}\ \textbf{Statement}$$

Fig. 1: Operational semantics of the range operator (*rng-map*)

Figure 1 presents the operational semantics of the range operator (*rng-map*) defined from Figure 4.3. The operator calculates the valid insert locations of the element given as its arguments. It shows how the range operator is interpreted when it is applied to the different elements (variables, expression, and statement) of TFA. We do not present any other operational semantics, since they are not fundamentally different from previous work [Schwartz *et al.*, 2010].

The rules in Figure 1 are read bottom to top, left to right. Given a range statement, we pattern-

match its argument to find the applicable rule. e.g., given the range statement `r(eax1 := 0x1 & eax0)`, we first match to the *Statement* rule. We then apply the computation given in the top of the rule, and if successful, a transition made from the original context (Σ) to the updated context (Σ'.) Σ is the only context we care for the operator, and it represents the instrumentation locations in the block being analyzed. For a BB with n instructions ($size(\Sigma) = n$), it allows $n + 1$ instrumentation locations ($[0, size(\Sigma)]$).

For *constant variables*, the range operator returns locations available in the entire block ($[0, size(\Sigma)]$). For *register variables*, it returns the range from the previous to the next definition ($[def(reg_{n-1}), def(reg_{n+1})]$.) In the case of *memory variables*, it looks for ranges where both the variables used in addressing and the memory variable itself are valid concurrently. The valid range for *expressions* and *statements* can be obtained by combining the ranges of their contained elements. Unlike other rules, the *Statement* rule updates the context (Σ) by inserting its input statement ($var ::= expr$) to a computed valid location (l). The algorithm to choose the insertion location is discussed in Section 4.3.3.

1.2 Theorems and proofs

1.2.1 Soundness Theorem

In this section, we provide the proof for Theorem 1 presented in Section 4.3.3. The efficiency and correctness of a typical live variable analysis is proven using a semi-lattice framework [1]. We apply the same methodology to guarantee the correctness of outer analysis. In our context, correctness means that our optimized code has the same effect as the original data tracking logic. While our analysis is almost identical to a typical live variable analysis, we only assume that an incomplete CFG is available.

Live variable analysis fits in a semi-lattice framework, which is employed to verify the safety and efficiency of an entire family of data flow problems. The data flow equations for the analysis can be stated directly in terms of $IN[B]$ and $OUT[B]$, which represent the set of variables live at the points immediately before and after block B. Also, we define

1. def_B as the set of variables *defined* (i.e., definitely assigned values) in B prior to any use of that variable in B

2. use_B as the set of variables, whose values may be used in B prior to any definition of the

variable.

The transfer functions relating def and use to the unknown IN and OUT are defined as:

$$IN[EXIT] = \phi \tag{1}$$

and for all basic blocks B other than $EXIT$,

$$IN[B] = use_B \cup (OUT[B] - def_B) \tag{2}$$

$$OUT[B] = \bigcup_{\textit{S successor of B}} IN[S] \tag{3}$$

Equation 1 is a boundary condition, which defines that no variables are live when the program exits. Equation 2 specifies that a variable coming into a block is live, either if it is defined before usage in that block, or if it is live when coming out of the block. The last equation states that a variable coming out of a block is live, only if it is live when coming into any of its successors. These definitions comprise the most significant element of the semi-lattice framework. The monotone property of the above transfer functions is an important factor that confirms the safety of the analysis.

Since we only assume an incomplete CFG is available, we show that the same algorithm can be applied to our analysis. We modified the live variable analysis by adding a conditional statement to Equation 3 which defines all variables as live, if an unknown successor block exists (e.g., due to an indirect branch). This in turn replaces Equation 3 with Equation 4.

$$OUT[B] = \bigcup_{\textit{S successor of B}} IN[S] \tag{4}$$

(For any unknown S, $IN[S] = U$)

By having this equation, we claim the following theorem:

Theorem 1. ***Soundness of outer analysis:*** *Live variable analysis with incomplete CFG converges and is safe.*

Proof. The algorithm for the analysis will terminate, as we always have a *finite* number of blocks. Also, the additional statement in Equation 4 does not have any effect on the *monotone* property of the transfer functions, as it imposes conditions only on input arguments, and not on the function itself. Thus, it follows from the definition that the modified analysis still fits in the semi-lattice framework. □

1.2.2 Efficiency theorem

In this section, we provide the proof for Theorem 2 presented in Section 4.3.4. The theorem states:

Theorem 2. ***Efficiency of the TFA optimization****: The TFA optimization always produces less, or an equal number of, tracking statements than the original representation, for any basic block.*

Proof. Note that *n()* returns the number of statements in taint-map and T, T', and T'' represent the taint-map data structures generated from the different stages of the TFA optimization.

T : the original taint-map directly translated from a given basic block B

T' : the copy propagation (substitution algorithm) applied to T

T'' : the range correction is applied to T'

then we want to show $n(T) \geq n(T'') \geq n(T')$.

An example taint-map data structure and its DAG representation is shown in Figures 4.2(b) and 4.4 respectively. Showing $n(T) \geq n(T')$ and $n(T'') \geq n(T')$ are trivial, as the substitution algorithm combines two or more statements into a single statement, and range correction splits a statement into two or more statements.

What we need to prove is $n(T) \geq n(T'')$. To show this, we use proof by contradiction. Let's assume that T'' is an optimal taint-map that does not have any range violations, and $n(T) < n(T'')$. WLOG, we can say that a taint-map can have statements such that $\exists s_0, s_1 \in T$ merged into $s'_0 \in T'$ by substitution algorithm, and s'_0 again split into $s''_0, s''_1, s''_2 \in T''$ by range correction. Then, we can have a new taint-map T''' such that $T''' = (T'' - s''_0, s''_1, s''_2) \cup s_0, s_1$. The newly created T''' does not have range violation and $n(T''') < n(T'')$. This contradicts our assumption that T'' is optimal taint-map with the minimal number of valid statements. □