UC Irvine

UC Irvine Electronic Theses and Dissertations

Title

Introspective Intrusion Detection

Permalink

https://escholarship.org/uc/item/5dt7v1ck

Author

Hawkins, Byron

Publication Date

2017

Copyright Information

This work is made available under the terms of a Creative Commons Attribution License, availalbe at https://creativecommons.org/licenses/by/4.0/

Peer reviewed|Thesis/dissertation

eScholarship.org Powered by the California Digital Library University of California

UNIVERSITY OF CALIFORNIA,
IRVINE

Introspective Intrusion Detection

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Computer Engineering

by

Byron Hawkins

Dissertation Committee:
Associate Professor Brian Demsky, Chair
Professor Michael Franz
Associate Professor Athina Markopoulou

2017

Portions of Chapters 1, 2, 5 © 2017 IEEE
Portions of Chapters 1, 2, 3 © 2016 ACM
Portions of Chapters 1, 4 © 2015 IEEE
All other materials © 2017 Byron Hawkins

DEDICATION

To the
International Justice Mission
for their enduring effort
to protect powerless individuals
from the malicious intrusions
of the real world.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

List of Algorithms

ACKNOWLEDGMENTS

I would like to thank:

- My parents for their continual support in all my life's endeavors, and especially for attending my presentation of ZenIDS at ICSE 2017 in Buenos Aires;
- My advisor Brian Demsky for envisioning a dynamic profiling approach to software security and giving me the opportunity to participate in its development as it gradually evolved into IID, and for making a dedicated personal effort on each of my conference submissions;
- My doctoral committee for focusing the direction and position of this dissertation;
- Michael Taylor for teaching me how to set up the expectations in a research presentation;
- Michael Franz for his advice about my academic career and future research possibilities;
- Fabrice Rastello and Erven Rohou for giving me the opportunity to pursue a career in research;
- Derek Bruening for mentoring my work on DynamoRIO and teaching me how to code in C;
- James Radigan for giving me the opportunity to implement a security technique for a major commercial product, and Ten Tzen, Shishir Sharma and Neeraj Singh for mentoring my work;
- Yong hun Eom for his friendship and for many lengthy discussions about everything;
- Per Larsen, Andrei Homescu, Fabian Gruber, Erick Lavoie, Peizhao Ou, Bin Xu and Rahmadi Trimananda for their insightful comments about my paper and poster drafts;
- Ian Krouse, the UCI music department, et l'école de musique à Grenoble who provided a venue for balancing this intense technical effort with the art of emotional expression;
- My Lord Jesus Christ for creating a universe filled with mysterious dichotomies, where truth dwells under the shadows of the license to choose, and to designate meaning.

This work would not have been possible without generous support from the National Science Foundation under award 1228992 and grants CCF-0846195, CCF-1217854, CNS-1228995, and CCF-1319786; and from the Center for Future Architectures Research (C-FAR), one of six centers of STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA.

I would like to thank the ACM SIGPLAN and IEEE for granting permission to include content in this dissertation that has been previously published in conference proceedings.

CURRICULUM VITAE

Byron Hawkins

EDUCATION

Doctor of Philosophy in Computer Engineering **2017**
University of California, Irvine *Irvine, California*

Master of Science in Computer Engineering **2014**
University of California, Irvine *Irvine, California*

Bachelor of Science in Computer Science **1996**
Pacific Lutheran University *Tacoma, Washington*

Bachelor of Musical Arts **1996**
Pacific Lutheran University *Tacoma, Washington*

PUBLICATIONS

ZenIDS: introspective intrusion detection for PHP applications **ICSE 2017**
Byron Hawkins and Brian Demsky
International Conference on Software Engineering

BlackBox: lightweight security monitoring for COTS binaries **CGO 2016**
Byron Hawkins, Brian Demsky, and Michael B. Taylor
International Symposium on Code Generation and Optimization

Optimizing Binary Translation of Dynamically Generated Code **CGO 2015**
Byron Hawkins, Brian Demsky, Derek Bruening, and Qin Zhao
International Symposium on Code Generation and Optimization

COMMERCIAL PROJECTS

Security enhancement for Microsoft Visual Studio `https://www.visualstudio.com`
Internship at Microsoft Corp. · Redmond, WA, USA · Fall 2016
Implemented and tested a new security feature for the backend code generator of the Visual Studio C++ compiler (details restricted by NDA).

Dr. Fuzz `http://drmemory.org/docs/page_fuzzer.html`
Internship at Google, Inc. · Cambridge, MA, USA · Summer 2015
Fuzz testing tool for x86 and ARM binaries based on DynamoRIO and Dr. Memory.

OPEN-SOURCE SOFTWARE

ZenIDS `https://github.com/uci-plrg/zen-ids`
PHP extension for introspective intrusion detection (includes augmented PHP 7 interpreter and data processing tools).

BlackBox `https://github.com/uci-plrg/blackbox`
Introspective intrusion detection framework for x86 binaries based on DynamoRIO (includes augmented runtime and data processing tools).

ABSTRACT OF THE DISSERTATION

Introspective Intrusion Detection

By

Byron Hawkins

Doctor of Philosophy in Computer Engineering

University of California, Irvine, 2017

Associate Professor Brian Demsky, Chair

Remote code execution (RCE) exploits are considered among the more destructive intrusions in today's software security landscape because they allow the adversary to execute arbitrary code with the permissions of the victim process. Today's leading software defense systems are able to eliminate significant portions of this attack surface, but many of these approaches such as control flow integrity (CFI) and intrusion detection systems (IDS) address a limited scope of attack vectors, while others like diversification have not received wide user adoption. Consequently, user applications remain vulnerable to RCE attacks. One limitation that is common among these approaches is an over-commitment to preventing the adversary's takeover strategy. Since the domain of potential attack vectors may be infinite, the adversary may always be able to devise a new takeover strategy that cannot be detected or prevented by any of the existing takeover-oriented defenses. After an attack, current monitoring systems often do not provide sufficient information about how the adversary took control or what vulnerability was compromised. Forensic tools can derive this information from an instance of the attack, but it may take multiple days or even weeks to isolate an instance for analysis.

Motivated by the need for a program monitor that is equally effective against unforeseen takeover strategies as well as an exploit payload, *Introspective Intrusion Detection* (IID) combines the advantages of traditional intrusion detection and CFI by distinguishing anomalies in execution without making absolute judgments about malicious intent. This approach begins with a profiling phase that distills observed execution paths into a compact data representation called a *trusted profile* that constitutes a “ground truth” of safe application behavior. When deployed for online monitoring, IID reports any deviation from the Trusted Profile as a potential intrusion. Software developers can benefit from IID as a complement to deployed defenses by gaining visibility into unforeseen malicious behaviors or evasive variations of known attacks. Debugging of field failures and error reports can also benefit from the deep introspective logging provided by IID. Where deployed software has ample performance headroom and sufficient technical support, IID can also be deployed as a comprehensive realtime monitor to detect advanced persistent threats (APT) and other evasive intrusions.

Experimental evaluation of IID prototypes for x86 binary executables and PHP applications show that this technique (a) identifies unforeseen takeover strategies along with exploit payloads, (b) accurately distinguishes benign anomalies from those associated with real attacks, (c) incurs low overhead with minimal false positives during normal execution of server applications such as Microsoft IIS and web applications such as WordPress, and (d) incurs moderate overhead for desktop applications such as Microsoft Office, Google Chrome and Adobe PDF Reader with a false positive rate suitable for expert analysis.

Chapter 1

Introduction

As software increasingly relies on dynamic components such as plugins, task pools and script interpreters, it becomes increasingly important to defend the inherent vulnerability of a program's dynamic control flow. Remote Code Execution (RCE) attacks typically manipulate the runtime decisions of a program to take control and launch arbitrary code with the permissions of the victim process. The increasing prevalence of dynamic program behaviors not only raises the probability that a structural flaw will expose decision points to malicious influence, it also gives the adversary more ways to acquire full control from any given manipulable state. While it is often possible to eliminate a program's vulnerabilities by patching the code, there are cases where the vulnerability is inherent to the design of the program. For example, the WordPress API for remote procedure calls (known as "XML RPC") is capable of issuing thousands of login attempts and circumvents the delay between invalid logins. Security experts agree this is a flaw in the design of WordPress because it allows brute-force password attacks—yet the WordPress development team refuses to patch the API, arguing that the attacks are an unfortunate abuse of a valid and necessary application feature. This suggests it may be possible for an implementation to be perfect—having the exact semantics intended by its design—and still exhibit vulnerabilities.

In this sense, dynamic software constructs can be a two-edged sword, both benefiting a project and at the same time potentially introducing vulnerabilities. Many organizations claim that the flexibility of dynamic programming languages such as of PHP can reduce time to market [2, 6, 157], yet those same features are common targets of attack (Section 5.2). User-defined types such as C++ classes can improve reliability and flexibility throughout the lifespan of a program [139, 158, 130], yet these often rely on runtime control flow constructs such as the vulnerable C++ dynamic dispatch tables [149]. Plugin architectures encourage community development, for example there were over 54,000 WordPress plugins as of August, 2016 [40], and Chrome extensions are not far behind [160], yet malicious plugins have attacked users of both platforms, and many benevolent plugins contain vulnerabilities that are frequently compromised (Section 5.5.1). Dynamic browser components such as Flash and JavaScript interpreters have enabled an entire application platform to operate online, yet they have also enabled entire new classes of exploits such as cross-site scripting (XSS) and drive-by-downloads.

What makes the security effort especially difficult is that the malicious control flow of an exploit often takes a similar form to the benevolent control flow of these dynamic features. For example, during startup of Microsoft Office 2013, a licensing module generates a small code fragment and links it dynamically using a `mov; ret` sequence, but this program behavior is structurally identical to many popular code injection exploits. Similarly, the Fibers API in Microsoft Windows implements a soft context switch by overwriting the CPU register state and executing a stack pivot, much like the escalation stage of a typical buffer overflow attack. Where it is difficult for even a security expert to make a *precise* and *universal* distinction between the program's intended control flow and adversarial manipulation, it will be even more challenging to apply a generic security policy that automatically protects programs from those attacks at runtime.

Intrusion Detection

An early security approach focuses on patterns in untrusted inputs and internal program events to distinguish potential intrusions. By defining a whitelist of normal activity and/or a blacklist of prohibited activity, unwanted program behaviors can be detected and thoroughly logged to improve forensic efforts, enabling temporary protections until software can be patched. But as applications have grown increasingly dynamic, it has been difficult to manually maintain an effective whitelist or blacklist. Research into automatically generating whitelist patterns or even n-grams and finite state automata have met with limited success. The inherent abstraction of these approaches provides slack for the adversary to craft behavioral disguises that artificially conform to statistical norms or functional models. Anomaly detection has only continued to find success outside of the software domain, for example in fraud detection and medical diagnosis. Much of the work in intrusion detection has been narrowed to specific programs for this reason, since generalized models of intrusion have not been effective in practice.

CFI

Focusing more specifically on program structure and the behavior of an exploit, techniques arose to obstruct the execution paths of attacks. For example, operating systems now commonly assign non-executable permissions for the memory pages holding the call stack to prevent an adversary from injecting code into a buffer overflow and executing it directly. In response, new exploits were developed that link together enough fragments of existing executable code to fabricate an artificial call to `VirtualProtect` (or similar) and grant executable permission to an injected payload. Based on an insight that this kind of code reuse attack leverages dynamic program constructs such as function pointers, researchers began investigating the potential for enforcing control flow integrity to deter adversarial abuse of existing program content. But this approach has been limited by the difficulty of determining a valid set of targets for any particular program branch. To improve runtime performance and ease offline analysis, most implementations group branches into categories and allow a branch to reach any destination that is valid for its category. While this increases the effort required to generate a successful exploit, many of today's CFI defenses can be defeated by sophisticated attacks. An additional limitation of CFI is that, once the adversary gains control, it cannot do anything further to hinder the adversary, and is also unable to provide any information about how the attack occurred or what subsequent actions were taken.

Section 1.1 presents a timeline of the major categories of intrusion detectors and CFI defenses, focusing on their strengths and limitations.

Introspective Intrusion Detection

In an effort to close the visibility gap in today's security platforms while defusing the surprises in tomorrow's exploits, Introspective Intrusion Detection defines trust locally in terms of internal control flow on a per-program basis and reports any untrusted edge as an anomaly. Deployment of an IID begins with a local profiling phase that captures an under-approximation of normal program behavior in the form of an uncompressed control flow graph (CFG), without path or context sensitivity and excluding frequency of edge traversal. When the IID runtime monitor is enabled, any deviation from this Trusted Profile is reported as a potential intrusion. This approach offers three unique advantages:

1. **Local Policies** are inferred from observed executions and can be manually adjusted, improving visibility into vulnerabilities that are difficult to define in generic terms, or are simply not universal to all deployments of the application, while avoiding laborious configuration.
2. **Fine-Grained Trust** minimizes the adversary's opportunities for mimicry and evasion.
3. **Pessimistic Evaluation** based on an under-approximation of trusted behavior makes no assumptions about the form of malicious control flow or how the adversary might invoke it.

There are also disadvantages inherent to dynamic profiling and runtime introspection that are not common either to intrusion detectors or to CFI defenses in general. Runtime overheads can be challenging on some platforms, and sufficient profiling coverage can be difficult to obtain without risking adversarial influence. Implementing an effective IID requires a clear understanding of its inherent advantages, along with careful engineering to prevent its challenges from outweighing these benefits.

Local Policies

While generic security policies can be simpler to enforce, for many important vulnerabilities it is easier to detect adversarial influence on a per-deployment basis. For example, consider two WordPress sites, where users of the first site never use the authentication command in XML RPC, but users of the second site use that feature regularly. An IID monitoring the first site will not trust the XML RPC authentication code, even if the site administrator has never heard of XML RPC and does not know that non-interactive logins are possible (this is probably true of most casual bloggers). Any HTTP request attempting a login over XML RPC to the first site will be logged as an anomaly, but if the same login request is sent to the second site, that IID will quietly trust it. Similar issues arise for plugins and themes, any of which may be welcome on one site yet forbidden by the administration of another site.

Fine-Grained Trust

When a defense technique relies on statistical abstractions or the approximations of simplified models, it is frequently discovered vulnerable to mimicry and evasion tactics. Because software components of any size often have fully reversible effects, there can be an infinite degree of slack between the individual behaviors of program components and any composite behavior. A statistical abstraction or an approximative model focuses on program behaviors occurring at some fixed level of granularity, usually for the purpose of improving runtime performance, or simplifying development and administrative effort. An adversary can take advantage of the slack between the security tool's chosen level of granularity and the composite behavior of the program, making the malicious behavior conform to the definition of secure behavior at the tool's level while performing an attack at the composite level.

Introspective Intrusion Detection is not immune to this tactic, since it is theoretically possible to implement any composite program behavior within the confines of any sufficiently large and diverse control flow graph. But in practice it has not been possible for an exploit to gain arbitrary code execution without either (a) executing an arbitrarily composed payload within the victim process, or (b) forking a new malicious process, both of which are easily detected by a well-configured IID. It has also not been possible to manipulate the arguments to a system call without adding edges to the control flow graph—except where the program suffers from unsanitized inputs, which is outside the scope of RCE exploits and is better addressed by other tools focusing on data-only attacks. Chapter 6 provides a comprehensive review of recently published attack strategies and discusses the potential for an RCE attack to fly under the radar of a well-configured IID. So, while conformity to the Trusted Profile does not provide any theoretical or scientific guarantee of security, it can be shown capable of detecting even the most sophisticated of known techniques for evading CFI defenses.

Pessimistic Evaluation

By limiting trust to the set of control flow paths observed in real executions of the application, IID maximizes visibility into the unexpected. Recent attacks have defeated promising CFI implementations by taking advantage of the definition of unsafe control flow. Similarly, attacks have compromised the simplified models of normal behavior in anomaly and intrusion detectors. These strategies for masking an attack are not available against IID because (a) it does not define malicious behavior at all, but simply reports everything untrusted as a potential intrusion, and (b) it does not compress the CFG or simplify it into a model. Even where the rate of control flow anomalies is too high for regular investigation, the information remains available should an attack be discovered later. Experiments reported in Chapter 3 show that ordinary usage of today's large and complex desktop applications under IID produces a manageable set of anomaly reports, even with a Trusted Profile generated from just one user (Table 3.3).

Section 1.2 discusses the advantages and challenges of Introspective Intrusion Detection in more detail, and Section 1.3 derives the security and usability goals of IID.

Usage Model

Introspective Intrusion Detection can be an effective tool for software developers, security analysts and in many cases the end users themselves. **Software engineers** can use IID to discover unexpected behaviors in their own programs, whether it is triggered by accident or by malicious intent. The majority of profiling can be done in a secure environment with existing infrastructure, such as automated and manual tests and sample executions for profile-guided compiler optimizations. **Security analysts** may explore malicious activity occurring in the wild, for example by deploying IID in a honey pot. **End-user** environments having ample performance headroom and sufficient technical support—whether IT staff or an astute blogger—can deploy IID for their own direct benefit. The IID blacklist can be used to mitigate observed program vulnerabilities (or those reported by the community or security analysts) until the vendor can release a patch. All of these use cases ideally leverage the Trusted Profile distributed by the vendor, and users can make adjustments to focus the IID logs according to their needs.

Section 1.4 continues this discussion of usage scenarios in more detail, along with profiling and associated security concerns such as advanced persistent threats. Section 1.5 previews two prototype implementations of IID: BlackBox for x86 binaries and ZenIDS for PHP web applications. Section 1.6 presents the contributions of this dissertation, and Section 1.7 outlines its organization.

1.1 Timeline of Important RCE Defenses

This section outlines the history of security techniques that focus on protecting a program from RCE exploits. An early response to adversarial manipulation that continues today is to simply find and fix program errors and exploitable vulnerabilities (Section 1.1.1). Despite the obvious effectiveness of this approach, at some point it became evident that developers would never be able to produce invincibly correct programs, so researchers began pursuing intrusion detection based on simplified statistical or structural models (Section 1.1.2). Throughout this dissertation, this approach will be referred to as *traditional intrusion detection* (TID), although this is not a standard name (or acronym). As the Internet matured, web applications came to rely increasingly on dynamically generated HTTP interfaces, while desktop applications—which were already highly dynamic—began to integrate Internet-facing components such as template and image browsers and integrated cloud storage systems. These additional dimensions of variability quickly overwhelmed TID, leading researchers to pursue a new approach called *control flow integrity* (CFI) based on the insight that attacks often change the low-level flow of execution within a program (Section 1.1.3). Diversification randomizes various components of a program to interfere with an attacker's expectations (Section 1.1.4).

Introspective Intrusion Detection attempts to combine the strengths of CFI and TID by focusing on low-level control flow (like CFI but unlike TID) while relying on an under-approximation of normal behavior (like TID but unlike CFI). This approach can broaden the detection capabilities over existing CFI approaches while maintaining compatibility with today's highly dynamic applications. Chapter 6 presents this body of related work in more detail, focusing on important defense techniques in detailed comparison and contrast with IID.

1.1.1 Bug Fixing

For the purpose of improving security, the techniques for finding and fixing bugs can be divided into two categories. The first examines a program for known kinds of errors and may suggest or even generate repairs. The second looks for potential exploits and may generate a POC. Orthogonal work detects program errors at runtime and takes corrective or protective action. Security is improved by eliminating the adversary's opportunities to invoke unexpected behavior.

Offline Error Detection focuses on unexpected program behaviors that arise from errors in the construction of the program, for example the notorious buffer overflow in which the program naïvely assumes that any index into a physically sequential data structure is probably valid for its current size. Myriad tools have been developed to detect and correct these bugs, both at compile time and dynamically at runtime, some requiring no developer intervention. When successful, these tools eliminate certain kinds of unexpected program behaviors, though it has proven difficult to eliminate all such bugs from complex programs, despite major commercial investment.

Exploit Discovery pursues subtle program bugs that may not conform to common patterns found by standard debugging techniques. Since these bugs are difficult to find, specialized tools focus specifically on detecting opportunities to manipulate program state. This approach is especially popular for dynamic programming languages where structural defenses like CFI are difficult to apply. Web applications also favor this approach because it can potentially detect second-order vulnerabilities that traverse persisted program state, which are transparent to traditional bug discovery techniques.

Online Error Detection continuously monitors program execution for typical symptoms of errors. For example, a stack canary occupies a slot following any buffer and can be checked at any time for corruption, which often indicates that a buffer write has overflowed its bounds. This approach tends to be limited by performance constraints, requiring relatively simple implementations that an adversary can work around. Stack canaries have been defeated by exploits that read the expected value prior to overflowing the buffer, then patch the corresponding slot in the malicious buffer to avoid detection.

Introspective Intrusion Detection often reveals program bugs that lead to control flow anomalies, although it may also detect anomalies not directly caused by an error in the construction of the program (for example, hardware malfunction).

1.1.2 Traditional Intrusion Detection

In one of the earliest techniques to be widely deployed in online systems, a monitor focuses on program input, continuously searching for patterns that are consistently associated with malicious intrusions. These approaches range in sophistication from manually maintained whitelists and blacklists to machine learning approaches that can automatically adapt to conditions after deployment. But this approach has faced many limitations. As programs become increasingly dynamic, it has become unrealistic to manually maintain intrusion detection filters. Mimicry tactics have increased the workload by randomizing the factors that are known to be easily detected. Yet recent attacks have also targeted machine learning directly, introducing specialized forms of mimicry that take advantage of weaknesses in the algorithm's feature distribution [8]. In general, intrusion detection based on program inputs has recently become a losing battle.

An orthogonal approach known as *anomaly detection* speculates that program activity not conforming to known norms may be malicious. This approach is often applied to program inputs, internal execution patterns, and statistical artifacts of observable program behaviors. While this approach can be effective for programs having simple internal structure or interfaces that rarely change, these techniques also do not scale well with program complexity. As interfaces and internal structure become increasingly dynamic, anomaly detection has largely been abandoned in favor of more precise techniques such as CFI.

Malicious Input Detectors focus on applications with relatively simple input interfaces, usually receiving data in a homogeneous format that uses a well-defined alphabet. Malicious inputs are defined on a per-application basis and can often be combined into categories for fast detection using finite state automata (FSA). As computing throughput increases, these are often implemented in specialized hardware or FPGA.

Introspective Anomaly Detectors apply FSA to internal program behavior, for example generating models of normal sequences of system calls for a particular program. A similar approach derives n-grams from known-safe executions. But these tools tend to be brittle, raising false positives on slight variations such as synonymous system calls that do not signify a semantic deviation. Dynamic events such as interrupts can also be difficult to handle. Conversely, adversaries have been able to implement malicious behavior that conforms to these simple models, making them ineffective against targeted attacks.

Statistical Anomaly Detectors typically rely on a set of expected proportional relationships and raise suspicion about outlying events. For example if a program typically outputs between 1KB and 10KB per invocation, a statistical anomaly might be detected for an invocation that generates 10MB. This approach is similarly susceptible to mimicry, for example an adversary can divide the 10MB payload into a thousand chunks and distribute them across a thousand statistically normal invocations. False positives are also common because programs frequently have special cases that do not conform well to norms.

Introspective Intrusion Detection defines a similar under-approximation of normal program behavior, but expands the scope to the entire program CFG, which adapts better to program complexity than the smaller FSA and n-gram models, and reduces opportunities for mimicry.

1.1.3 Control Flow Integrity

Programs use branches with dynamically specified targets to facilitate flexible semantics, but for any given branch, the program rarely intends for it to reach any possible target. Since attacks can manipulate the branch argument to reach targets that were never intended, CFI protects a program by imposing constraints on branch targets that limit adversarial opportunity. The effectiveness of a CFI approach is largely a function of its accuracy in identifying branch targets that the program intends to use. Other factors include the vulnerability of the CFI mechanism itself, which can be the victim of targeted attacks that disable the CFI constraints, and the inconvenience costs of the CFI tool.

Branch Target Accuracy is largely determined by the mechanism used to derive the set of valid targets. The conventional approach for forward edges (i.e., those that push a stack frame or are stack neutral) is to perform a static analysis on the source code to discover, for each branch, a set of target constraints implied or specified by associated program elements. This approach typically approximates structural targets such as virtual method calls, and may derive no constraints for function pointers such as callbacks. For convenience when source code is not available on a compiled platform, some CFI techniques analyze branches directly in the compiled binary. These approaches have much less information and typically make much broader (weaker) approximations. Based on the insight that the adversary cannot manipulate a branch argument until after the program has placed it into a register or memory slot, dynamic CFI systems protect the program's branch targets at runtime without attempting to determine which targets are safe. One approach encrypts the target wherever it becomes vulnerable and decrypts it immediately prior to the branch, while another approach puts all branch targets into a separate stack. Dynamic approaches are the most accurate, at the cost of higher overhead, while static analysis tends to apply weaker constraints but very efficiently. Most CFI systems protect backward edges with a shadow stack or some other dynamic mechanism at near-perfect accuracy, though some have attempted to use the x86 *last branch record* (LBR), and others do not address backward edges at all.

CFI Self-Defense To apply branch constraints at runtime, most CFI systems maintain internal state, either throughout the execution or during certain program events. An attack targeting the CFI defense may be able to control its internal state, for example if it finds register values spilled on the stack. Where CFI uses encryption, it can be vulnerable to poisoning attacks that collect correctly encrypted branch targets for later abuse. Safe stacks and shadow stacks can be found using side channel leaks and manipulated or disabled. Constraints built into program constructs such as virtual dispatch tables can be manipulated with counterfeit objects. Any attempt to use the LBR is easily defeated with crafted gadgets. Attacks have been proposed and often proven against most of the important CFI systems developed in research and many industrial implementations as well.

Inconvenience Cost The primary concern of most CFI implementations is to limit runtime and memory overhead. Compiler-based techniques may increase code size, but this is rarely a significant cost. Safe stacks and shadow stacks may consume some of the preciously few slots available in *thread local storage* (TLS), or may occupy the TLS register itself. Encryption-based CFI typically requires a dedicated register, which is substantially unrealistic on some platforms like 32-bit x86 where general registers are already severely lacking. Some CFI tools improve branch target accuracy by dynamically profiling the program, which is rarely a simple and straight-forward task. A common cost that is often neglected by all kinds of CFI systems is compatibility with programs and operating systems, ranging from unnecessary false positives for uncommon program structures to—all too commonly—a total inability to function on the Windows platform.

Introspective Intrusion Detection may be the first CFI approach to under-approximate the set of intended targets at each branch. As such it is not able to enforce CFI at runtime, since it has an inherent tendency towards false positives that would prevent programs from executing correctly. It is possible for the Trusted Profile to contain unintended branch targets, mainly in the case of adversarial influence during profiling. Although the Trusted Profile is loaded into a read-only `mmap`, it may be possible for an adversary to pollute it online, either by writing to the `mmap` region or by modifying values the IID copied from it. Various other IID data structures may also be leaked and/or manipulated by the adversary, though an open question is whether the adversary can gain this control without diverging from the Trusted Profile in the process. Timing side channels may be able to reveal some of the Trusted Profile contents, though the same open question applies. Attacks on the IID runtime are less feasible on managed platforms where arbitrary read/write access is much more difficult to acquire. The inconvenience costs of IID are typically higher than for other CFI approaches, including higher runtime and memory overhead, and the administrative effort of profiling.

1.1.4 Diversification

To gain full control of a program through unexpected behavior, the adversary often leverages detailed knowledge of the program's internal construction, intricately linking program fragments to compose any possible context switch into the malicious payload. The key insight of *diversification* is that small, randomized changes to the program can interfere with these intricate exploits, causing the vast majority of attempts to fail. Compiler-based diversification tools randomize the generated code such that that one program instance is rarely vulnerable to an exploit that works perfectly on another instance. But diversification is most effective against attacks that leverage differences between a compiled executable and the original source semantics. It has been more difficult to diversify interpreted language platforms because it is challenging to diversify source code. Diversification also does not attempt to protect vulnerabilities caused by a programmer's over-approximations.

Introspective Intrusion Detection is based on the binary translator DynamoRIO and as such is capable of diversifying the target program as it executes. If this were done selectively, low overhead could be achieved. It is also possible to apply runtime diversification to the Trusted Profile as a counter to adversarial influence.

1.2 Theoretical Advantages and Challenges of IID

This section discusses the essential design and implementation factors that can make an IID more or less successful. To make a clear connection with the experimental evaluation of the IID prototypes, Section 1.3 derives a set of specific goals for IID on the basis of these enumerated advantages and challenges. To assist software engineers in designing an IID for a new platform, labels A_{tag} and C_{tag} are assigned to the factors that can potentially be maximized or minimized (respectively) to improve the quality of an IID. Labels are referenced throughout the text for continuity. There are other important aspects of IID, but these are less sensitive to design decisions and do not require careful consideration at each step of the design process; these factors include:

- **Minimal Branch Target Classes**: Since indirect edges in the Trusted Profile are discovered dynamically, the set of trusted targets for each branch is even smaller than a perfect static analysis (modulo context sensitivity, which is possible as an IID extension).
- **Integrated Platform Support**: Many programs host additional runtime environments such as a JavaScript interpreter, which an IID can represent as an abstraction (Section 2.1).
- **Multiple Platform Support**: An IID can be created for any runtime, including native execution environments and script interpreters.
- **Dynamically Generated Code Support**: The runtime component of IID is able to monitor both statically compiled code and dynamically generated code in much the same way.
- **Road Ready**: An IID can successfully monitor today's popular applications on commodity hardware, including Google Chrome, Microsoft Office, and WordPress. Unlike most important CFI implementations developed in research, the IID prototypes are not showroom trophies. The BlackBox implementation was verified by independent third-party reviewers, and the reader is invited to take it for a test drive on their favorite Windows programs today.

Before looking at each advantage and challenge of IID in detail, it is important to clearly establish what it means to "detect" an exploit. Certainly a pure control flow log would, in some sense, detect every possible attack, since it is not possible to attack a program that is not running. But this is not a very useful sense of detection, because no security analyst could comprehend such a massive log (Table 3.3). On the other hand, it is not necessary for the IID log to be 100% free of false positives, especially where anomalies are ranked by estimated suspicion (Section 3.5.2). For the purposes of evaluation in this dissertation (Sections 3.6 and 5.5), the notion that an IID "detects" an exploit indicates that (a) it logs at least one control flow edge that is foreign to the monitored program and essential to the attack, and (b) the log clearly indicates something suspicious happened, even if other benign anomalies are simultaneously occurring and there are not yet any externally observable symptoms of an attack.

> Advantage $A_{\textbf{wpm}}$: **w**hole **p**rocess **m**onitor
>
> An IID monitors every control flow edge and process fork throughout the entire process, including unforeseen takeover strategies and the malicious payload.

The central advantage of IID that should be maximized by any successful implementation is that it monitors the whole process. This includes the temporal matter of continuously monitoring all program threads (or whatever instruction flows may exist on the platform), it also includes the simplicity of the monitoring algorithm. By observing program control flow in the simplest possible terms, an IID sees as much of the program behavior as possible, minimizing interference from pre-determined notions of "normal" or "safe" execution.

Many RCE defense techniques—including most CFI approaches—focus specifically on a set of anticipated adversarial takeover strategies, and can do nothing to deter the exploit payload once it gets started. An IID continuously monitors execution of the entire process, including unforeseen takeover strategies and malicious payloads. This does not guarantee that every possible attack will appear in the IID logs, but in practice it imposes significant limitations on the adversary. While strict reuse exploits such as Control Flow Bending (CFB) [21] and Control Jujutsu [54] are occasionally able to invoke one or two existing system calls in a large program without introducing foreign control flow edges, they are not able to arbitrarily invoke any system call used by the program with arbitrary arguments. Outside of strict reuse exploits, an IID will log at least one anomaly during the attack, since by definition it introduces at least one control flow edge that the application never uses.

Furthermore, should advances in exploits enable them to arbitrarily control system calls within the confines of strict code reuse, it is possible to extend IID with context sensitivity. For example, the algorithm for identifying small fragments of dynamically generated code (DGC) in BlackBox is purely context sensitive, up to the bounds of the DGC fragment. While the overhead of this feature is particularly high (since the applicable DGC fragments are small and rarely require validation), a more limited context history could be implemented using bitwise operations on a reserved register [50] with minimal runtime cost.

> Challenge $C_{\textbf{perf}}$: maintaining sufficient **perf**ormance
>
> ---
>
> Program introspection often increases runtime overhead and development cost for hardware-based execution platforms.

This leads directly to the most significant challenge for IID: it is necessary to have full introspection into the control flow of every program thread at all times. For platforms where application control flow is directed by software components, such as script interpreters, this is relatively straight-forward. But for hardware-based platforms such as native executables, gaining introspection into the running process is complicated and usually performance intensive.

> Advantage $A_{\textbf{log}}$: precise **log** content
>
> ---
>
> Runtime introspection enables IID to log precise control flow events that represent suspicious program behavior.

The cost of introspection, however, also brings with it some benefit. When an important anomaly does occur in the execution of a deployed program, an IID can begin generating a detailed trace of program behavior. The trace can be as simple as control flow branches taken on the anomalous thread, or as complex as a complete system call trace including a deep copy of the arguments. An IID is able to log this essential information when any unexpected control flow occurs, even if that control flow cannot be correlated to a recognized attack or a known vulnerability.

The forensic analysis process can be very difficult and time consuming when the attacked system does not provide detailed information about the adversary's actions. Perhaps more importantly, to prevent recurrence of the same attack, security practitioners and developers need to understand the vulnerability and how the exploit took advantage of it. But without a control flow log, this may require a complex process similar to reverse engineering, based on whatever small fragments of related information are available [189]. Even if artifacts such as packed malware are available, the analysis often relies on dedicated tools and expertise. Similarly, administrators may need to determine which systems were affected, for example to know whether confidential information may have been exposed. But without a system call trace, it may be impossible to determine what data was accessed by the adversary.

Of all the defense techniques listed in Section 1.1, none of them is capable of logging information about an execution that was not detected as an attack, and most of them are not designed to perform any logging at all. Conversely, the size and complexity of today's software makes it unrealistic to continuously log application control flow or system call activity. By limiting log output until an anomaly is observed, IID is able to report highly detailed information about suspicious program behavior without exceeding feasible log capacity or overwhelming security analysts.

Challenge $C_{\textbf{prof}}$: sufficient and valid **prof**iling coverage

Generating and maintaining a sufficiently complete and valid Trusted Profile is necessary for accurate logging but may require effort and expertise.

To accurately distinguish between ordinary program execution and a potentially malicious anomaly, an IID deployment relies on the validity and completeness of its Trusted Profile. If it is too sparse, the IID will flood the log with anomalies and it will be impossible for any human to isolate malicious behaviors. Conversely, if the Trusted Profile becomes polluted through haphazard static analysis or adversarial influence, the IID may trust some exploits and elide logging for them entirely.

For major commercial software, vendors may be able to provide a sufficient and valid Trusted Profile that is suitable for all usage scenarios. It may be more convenient to simply profile users in their usual—vulnerable—runtime environment, but this does not necessarily produce an ideal representation of the monitored program's normal behavior. For example, a web application monitored while receiving live traffic will likely encounter malicious HTTP requests (on average, over 25% of traffic to our research lab's WordPress homepage is malicious). If the application has been properly configured and maintained, it should handle those requests safely, but the question is whether to include those intrusion-handling control flow paths in the Trusted Profile. Some users may not be interested in reports of trivial hacks such as invalid logins, while others may prefer all malicious requests to be logged. Conversely, profiling only unit tests or other automated functionality may lead to increased false positives.

There are several techniques that can improve the validity and completeness of the Trusted Profile, such as program comprehension (Section 6.6.1) and symbolic execution (Section 6.6.2). But this research focuses instead on optimizing the design of the IID components related to profiling. For example, the granularity of the execution traces is chosen for each IID implementation separately, on a per-platform basis, and as shown in Sections 3.3.1 and 5.2.1, it can have a tremendous impact on false positives and false negatives.

> Advantage $A_{\textbf{loc}}$: **loc**al policy control
>
> ---
>
> Users may benefit from having direct control over a deployment-specific Trusted Profile of their application.

While it can be challenging to produce an effective Trusted Profile, there are advantages to having local control over its contents. Consider two deployments of the same application, each with a distinct user group and distinct usage patterns. In the first deployment, a certain feature may be rarely used and pose a high risk to the organization. This group's administrator may prefer to omit the feature from the Trusted Profile so that every access to the feature is thoroughly logged. At the same time, the second group may make regular use of the feature, and it may not pose any risk to them. That group will likely benefit from having the feature in its Trusted Profile. Customizations like this can be difficult to implement for a tool that enforces generic and systematic policies, yet are easily managed with a Trusted Profile that contains an intuitive representation of execution traces. Organizations concerned about advanced persistent threats may wish to diversify the Trusted Profile at edges having suspicious statistical characteristics, and A_{loc} makes this relatively simple.

Challenge $C_{\textbf{adm}}$: **adm**inistrative overhead

Deploying an application with IID requires additional administrative effort and some basic security knowledge.

Even when a sufficiently valid and complete Trusted Profile has been established for a particular application, there is additional effort required to deploy the IID. Both of our prototype implementations require installation of special runtimes. In addition, the Trusted Profile must be deployed in a way that is protected from adversarial tampering. System administrators or end users will need to monitor the logs to determine whether intrusions have occurred. In contrast, many of the existing security techniques discussed in Section 1.1 require no additional effort at the deployment site, for example the bug fixing approaches or the compiler enhancements.

Advantage $A_{\textbf{acc}}$: user-**acc**essible policy

The IID runtime makes it possible for users of all levels of sophistication to manage accurate, low-level security policies.

Although these aspects of IID may seem like a burden, software users at all levels of sophistication are becoming more aware of security issues and more directly involved in the protection of their applications and data. For example, WordPress is the world's most widely deployed web application, and Google Analytics shows that the search term "WordPress Security" has doubled in popularity every 4 years since the application was released in 2004. In contrast, similar searches like "WordPress Themes" and "WordPress Plugins" peaked around 2010 and have steadily declined to less than half their original volume [68, 69, 67]. One of the common themes among the results of a "WordPress Security" search are that every site is potentially vulnerable—even ordinary blogs—and that no security technique is invincible. These trends suggest that the number of users who are motivated to deploy an IID is only likely to increase.

One feature that may especially draw attention is the IID blacklist, which can terminate execution when known-malicious control flow is observed. This makes it possible to reliably prevent recurrence of a newly observed RCE exploit before the vendor of the vulnerable component releases a security patch. It only takes one security expert to identify a new exploit and publish the corresponding blacklist entries. At the IID user's site, a daemon can be configured to receive blacklist entries from trusted experts and automatically install them.

It is also possible for relatively inexperienced users to maintain their own trust policies on the basis of log entries. As discussed in A_{log}, an IID anomaly report specifies the untrusted control flow edge, which is the only identifier needed for either a Trusted Profile entry or a blacklist entry. For convenience, an IID tool can assist the user in designating either a log entry or a manually specified edge as trusted or untrusted.

Challenge $C_{\textbf{impl}}$: cost-effective **impl**ementation

An IID developer faces a trade-off between usability of the runtime and the maintenance cost of its code.

Our experience in developing the IID prototypes, along with our experimental results, indicate that industrial versions of these runtimes could probably be produced and maintained by a relatively small group of developers. But this can still represent a significant cost, especially considering how much security expertise is required, and the increasing market demand for any kind of expertise in software development.

The maintenance effort may be higher for some platforms than others. A potential worst case is an IID for a browser's JavaScript engine, since the browser as a whole would likely be updated several times per week. Developers would be required to test the IID after each update, and minor revisions could be frequent.

In contrast, the traditional intrusion detectors based on pattern matching have little contact with the host platform and could likely be maintained independently. For example, optimization of a pattern matching detector could go so far as to leverage specialized hardware, such as an FPGA, without raising concerns about compatibility with the monitored applications or the host platform. Conversely, the optimizations in our IID prototypes are tightly dependent on the internal structure of the host runtime, the operating system, and even the ISA of the processor.

1.3 Goals of Introspective Intrusion Detection

The purpose of this research is to show that an implementation of Introspective Intrusion Detection:

- Improves exploit detection over the state of the art in important ways.
- Complements orthogonal defense system by providing a thorough control flow log of any intrusion that other defenses fail to prevent.
- Provides usable control flow logs for real-world applications.
- Meets application performance expectations under normal usage conditions.
- Requires a reasonable development effort with a reasonable degree of skill and expertise.

The theoretical advantages of IID presented in Section 1.2 indicate that the approach can potentially accomplish its purpose, but that implementations will require careful resolution of the challenges inherent to the IID design. The remainder of this dissertation focuses on this simple question: can Introspective Intrusion Detection deliver what it theoretically promises? To guide this investigation, this section presents a set of concrete goals that should be met for an implementation of IID to be a success. Each goal is formulated as some combination of maximizing the theoretical IID advantages A_* and minimizing the anticipated IID challenges C_*. The purpose of these references is to (a) focus the rationale for these goals, and (b) provide a more complete basis for judging the effectiveness of the IID prototypes.

Goals presented with no corresponding challenges would in theory be effortlessly accomplished on an ideal platform—i.e., one with unlimited hardware resources where applications are perfectly structured for IID monitoring purposes. Chapter 2 focuses specifically on this scenario as a way of isolating the core components of IID from real-world complications. Section 2.1 separately describes the kind of obstacles that are commonly raised by today's popular application platforms.

For the purposes of the implementation guide (Appendix B), these goals are listed in Table A.1 in the manner of a UML requirements diagram, along with the lists of related advantages, challenges and snares. This can be a helpful reference when designing an IID for a particular platform. By understanding the purpose behind the goals, the essential issues for each core component of the new implementation can largely be determined and addressed in advance.

Detection

Advantage A_{wpm} can be taken to its extreme by deploying an IID with an empty Trusted Profile, in which case it naturally detects every possible exploit. But in this configuration it can generate billions of anomaly reports per hour (Table 3.3). Challenge C_{prof} can also be taken to its extreme by profiling every possible user input, or employing symbolic execution to profile every possible control flow path under type compliant data values. This is also not ideal for detecting exploits because, unless the program is perfectly constructed, some of those paths may be exploitable and should not be trusted. The following three goals provide criteria for striking a balance between A_{wpm} and C_{prof} such that the IID logs focus on the most useful cross-section of program behavior.

Goal $G_{\mathbf{kx}}$: detect **k**nown e**x**ploits

An IID should detect known RCE exploit against real-world applications without relying on predefined knowledge of its takeover strategy.

One simple way to evaluate the balance between A_{wpm} and C_{prof} that is achievable by an IID implementation (or a particular deployment) is to test it with known exploits. If these are not detected, it definitely can't be reliable for new exploits.

Goal $G_{\mathbf{rx}}$: detect e**x**ploits developed in **r**esearch

An IID should detect sophisticated RCE exploits developed in research.

While it may be difficult to obtain working instances of the advanced attack vectors explored in security research, it is always possible to hypothetically trace an exploit against an IID implementation with a specific Trusted Profile. Examples are presented in Sections 3.4.2 and 3.5.1.

> Goal $G_{\mathbf{wx}}$: detect e**x**ploits in the **w**ild
>
> ---
>
> An IID should detect RCE exploits occurring in the wild.

After an IID has met goals G_{kx} and G_{rx}, the real test for tuning A_{wpm} vs. C_{prof} is to monitor a program facing untrusted inputs. The inputs should be recorded for manual analysis of potential false negatives.

Usability

While the majority of technical discussion will focus on security challenges, implementation details and prototype evaluation, it is also essential for any security tool to be reasonably usable. Even if the IID is used exclusively by professionals, people in general have more important things to do than struggle with confusing or cumbersome software.

> Goal $G_{\mathbf{dnd}}$: **d**o **n**ot **d**isturb
>
> ---
>
> Given a reasonable effort to generate a valid Trusted Profile, spurious anomalies should rarely or never be reported.

The logging capabilities A_{log} are similarly dependent on a balance with C_{prof}. This goal focuses on the log itself, to make its usability more of a goal and less of an accuracy metric. For example, where thorough profiling is not available, a post-processing phase can significantly reduce or at least simplify noise in the anomaly reports. It may be possible to group anomalies into equivalence classes without loss of significant detail, either by retaining singleton members of each class, or by presenting one representative member per class in a user interface that hides remaining anomalies in each class under an expander widget. Techniques such as data mining and machine learning have also been applied on IDS in general to detect and remove false positives [137, 115, 119].

Goal $G_{\textbf{mod}}$: **mod**ify the Trusted Profile or blacklist

The user should be able to modify the IID security policy, for example by pasting a log entry into either the Trusted Profile or the blacklist.

The IID design presents an opportunity to combine A_{loc}, having a Trusted Profile and blacklist defined locally to each installation, with A_{acc} that makes policies accessible to the user. To maximize this opportunity, the IID tools should provide a fluid interface between log entries and the Trusted Profile and blacklist, allowing users to build up policies on the basis of past events. This goal adds another dimension to A_{log}, focusing on the fields within a particular entry and also the format, which can significantly affect the tooling effort. These advantages are all limited by C_{prof}, since manual access to local policies can only be valuable for reasonably concise logs.

Goal $G_{\textbf{dbg}}$: provide effective **d**e**b**u**g**ging assistance

The IID runtime can help software developers and security analysts perform forensic analysis of exploits and debug related program errors.

This goal expands the focus of G_{*x} beyond detection to matters of information gathering and reporting. Where the forensic effort following an exploit may only require information local to the anomaly, a debugging effort is likely to expand the scope of A_{log} to include more information from more sources, such as stack traces of other threads, contents of memory, register states, filesystem contents, and so on. Though often it is not possible to generically determine a selection of log content that is optimal for all usage scenarios, A_{acc} provides opportunities for the IID designer to make log output tunable for the intended use cases. More developer-friendly opportunities should also be considered, such as integrating an interactive debugging agent into an IID (if it is deployed primarily for debugging purposes, not in a security sensitive scenario). Evaluating this goal may involve some element of user study, either with members of the IID development team or perhaps beta customers.

> Goal $G_{\mathbf{blk}}$: effective **bl**ac**k**list design and implementation
>
> ---
>
> All aspects of an IID should be carefully considered in the design of blacklist features and the implementation of blacklist operations.

Evaluating the blacklist design can be more complex than just testing that it functions correctly. It includes technicalities such as applying the blacklist action promptly at the moment the prohibited control flow is observed, along with usability concerns such as overlap between the Trusted Profile and blacklist. A_{loc} can become a burden where large numbers of IID instances share the same policy. Forensics procedures may in some cases conflict with A_{acc}, perhaps requiring a timestamped version history to make previous blacklist state available at any future time. The breadth of IID usage requirements and functional expectations should be considered with respect to G_{blk}.

Cost Effectiveness

> Goal $G_{\mathbf{lite}}$: runtime overhead should be **lite**
>
> ---
>
> IID runtime overhead should not drastically change the user experience of the monitored program or introduce esoteric hardware requirements.

Runtime efficiency is essential for any security tool. But this goal also focuses on finding a realistic balance between C_{perf} and the development costs C_{impl}, which may already be significant before performance concerns are addressed. Usable performance should not rely on rocket science.

> Goal $G_{\mathbf{dep}}$: user **dep**loyable
>
> ---
>
> It should be possible for non-expert users to securely deploy the IID without having direct assistance from an expert.

As discussed in C_{adm}, there is already more administrative involvement in IID than many security applications, especially in comparison to automated CFI tools. This user effort can largely be mitigated with effective installers that are well maintained and consider platform and environment details in advance.

> Goal $G_{\textbf{dev}}$: reasonable **dev**elopment cost
>
> ---
>
> A successful IID should maintain a cost/benefit ratio that is practical, even for initial deployments of the technology.

For an IID design and implementation to be successful, all the goals of IID need to be balanced with the corresponding cost of development and maintenance. As shown in C_{impl}, it is challenging to build an IID in the first place, so its durability will depend on minimizing extremes of complication that may impede future development or even become impossible to maintain.

1.4 Usage Model

The usage model for IID mainly targets software engineers, including product developers who focus on security related debugging, application administrators seeking a reliable complement to installed defense systems, and security analysts whose professional function is to understand current threats and recommend protective measures. But throughout computing technology there is a trend towards end-user assimilation of increasingly technical devices and programs. For example, landline telephones are affordable, effortless to use, and offer highly reliable service—yet the majority of today's personal phone calls in the United States are made on complicated cellular devices that are expensive to own and operate, technically challenging to manage, and have a very high rate of service interference or even complete failure [142]. Anticipating the potential for end users to consider IID as a kind of Anti-Virus 2.0, the IID goals G_{mod}, G_{blk} and G_{dep} focus on this secondary usage scenario of IID.

1.4.1 Forensics

The U.S. National Institute of Standard and Technology (NIST) publishes a Computer Security Incident Handling Guide [29] that explains:

> For many organizations, the most challenging part of the incident response process is accurately detecting and assessing possible incidents—determining whether an incident has occurred and, if so, the type, extent, and magnitude of the problem.

As first steps in responding to an intrusion, the guide recommends determining:

D1 “which networks, systems, or applications are affected”

D2 “who or what originated the incident”

D3 “how the incident is occurring (e.g., what tools or attack methods are being used, what vulnerabilities are being exploited)”

This concisely describes the kind of security procedures that Introspective Intrusion Detection is primarily designed to assist. Goal G_{wx} addresses D1 by detecting unexpected and unforeseen exploits, including failed exploits in many cases. It can also provide assurance that a certain anomaly has not occurred on a particular system—if the malicious control flow paths are known, and the Trusted Profile for a particular program does not (erroneously) contain all of those paths, then the absence of anomaly reports provides strong evidence that the malicious behavior did not occur in that program. Goal G_{log} can be indispensable in determining D2 by generating a thorough trace of system calls and other essential program behavior after an intrusion is detected. The anomaly report itself can provide essential information in determining D3, revealing the low-level actions taken by software under malicious influence.

The form factor of an IID may not always seem ideal for deployment at end-user sites, especially considering its profiling requirement and potential for false positives when program behaviors differ for benign reasons such as environmental changes. However, the NIST also publishes a guide for Intrusion Detection and Prevention Systems (IDPS) [147] that recommends host-based monitoring for improved accuracy and specifically anticipates the challenges of local profiling:

- "**Profile Networks and Systems.** Profiling is measuring the characteristics of expected activity so that changes to it can be more easily identified."
- "**Understand Normal Behaviors.** Incident response team members should study networks, systems, and applications to understand what their normal behavior is so that abnormal behavior can be recognized more easily."

An IID effectively automates this process within the domain of program execution. It is not feasible for users to manually introspect program control flow, but an IID can build an effective profile of a program and report digressions as they occur. The guide cautions administrators that "the total number of indicators may be thousands or millions a day", and recommends to make a plan for maintaining accurate profiles to avoid floods of false indicators. Yet experiments with IID under relatively sparse profiling data indicate a much lower rate of spurious anomaly reports, ranging from a few dozen per hour for large desktop applications (Table 3.3) to just 3 unique false positives while monitoring a WordPress website for an entire year (Section 5.5.2).

1.4.2 Advanced Persistent Threats

A summit of major technology executives in 2011, including RSA[1] and TechAmerica, reports in one of its key findings that:

> The number-one advanced persistent threat (APT) attack vector is now not technology, but social engineering. Furthermore, security is no longer about trying to keep all intruders outside of the network perimeter, but rather acknowledging that security today involves living in a state of constant compromise.

Eddie Schwartz, CSO of RSA, recommended:

> A more realistic [approach] would be accepting the fact that you live in a world of compromise, and understanding that you have to work in that world, and work in a mode of triage, instead of constantly trying to push back hordes at the gate.

A major concern at the summit was "targeted malware that is, in some cases, just hours old", making it nearly impossible for automated defenses to keep pace [151]. Two defenses focusing on APT show how a systematic approach involving personnel throughout an organization can gain a resource advantage over the adversary and eventually overcome the threat. While an IID is a potential target of APT attacks, especially where the adversary may attempt to pollute IID profiling, it is also possible for an IID to play a key role in important defense strategies.

Kill Chains

A model of attack strategy that applies to many domains including military operations is called a *kill chain* [81] because it is composed of attack stages that follow a dependency sequence, such

[1] https://www.rsa.com

that failure at any link in the chain will most likely interrupt the entire attack. This strategy for defending against APT is based on two important insights:

- The adversary is only willing to make a limited investment in the attack, and has limited resources in comparison to the lengthy duration of the attack, which may span multiple years.
- For the intrusion to be cost effective, the adversary relies on reusing or repurposing a significant majority of the necessary tactics, because creating and perfecting new tactics quickly becomes prohibitively expensive.

The procedure for this defense is to model the attack as a sequence of seven phases, and for each phase determine how the adversary accomplishes the sub-goal and how that attack vector can be defeated. An IID can potentially detect adversarial actions in each of the final four phases:

- *Exploitation*: is the primary focus of IID, where the adversary has delivered a malicious input such as a PDF or Word document that causes unexpected behavior in the corresponding application leading to remote control of the process. Since detection focuses on low-level program behavior, it can be successful even if personnel inside the organization have been deceived or enticed through social engineering to participate in the attack.
- *Installation*: to the filesystem can be detected by the IID that was monitoring exploited program, since according to A_{wpm} an IID continues to monitor after anomalies have been observed. Installation within a long-running process can also be detected if an IID is monitoring that process.
- *Command and Control*: from within a known process can be detected by an IID. Even if the controller forks subprocesses for communication purposes, the default behavior of an IID is to monitor the execution of all forked processes, including unknown programs.
- *Actions on Objectives*: effectively comprises the APT payload, which is detectable by the key IID advantage A_{wpm}.

The particular advantage of this seven-stage process is to focus attention on the components of the attack that have come to be taken for granted. While it can be important to detect zero-day components in the kill chain, these are not necessarily essential. The author points out that it can be more effective to identify reused attack components, because defeating any of those will force the adversary into additional ingenuity, which raises the cost of the attack.

Three aspects of IID can be especially valuable in a defense that is organized around kill chains.

- An IID can detect zero-day components, raising awareness of the attack early and potentially indicating how to defeat it.
- Where social engineering gives the adversary a point of entry into the system, an IID can detect resulting changes in program behavior—even if those changes are not directly correlated to compromised resources or program errors.
- When defenders become aware of one attack stage and need information about others, they can install additional IID instances and more carefully analyze anomaly reports to detect adversarial activity where it may not have been immediately obvious.

Where APT relies on secrecy to maintain its presence, the accuracy and continuous observation of an IID can reveal stealthy attacker operations.

Dynamic Defense

One of the most difficult complications of APT is the participation of insiders, who occupy a seemingly invincible position when they have authorized accessed to crucial system resources. This dynamic defense strategy models the attack as a two-layer differential game, where one layer focuses on attacker vs. defender, and the second focuses insider vs. defender. In a highly mathematical analysis of corresponding cost models, it is revealed that a significant factor for insider participation is the risk of being caught, which increases significantly with defender vigilance. Contrary to intuition, the model shows that the defender can reach an equilibrium against the attacker while expending comparable resources, such that the cooperation of insiders leads to no net advantage for the attacker [79].

This is a fundamental insight for the role of IID in a world where APT threats are steadily increasing and have become pervasive in high-value organizations. An IID excels at detecting unexpected behavior in programs—even if the behavior is normal for the program in a technical sense. The more effort the defenders invest in accurate profiling and log analysis, the more effective an IID can be in revealing the subtle moves of stealthy insiders. Even if the IID becomes a target of insider manipulation, it preserves evidence of tampering. For example, if an insider deliberately pollutes the Trusted Profile with control flow edges corresponding to the execution of malicious agents, that Trusted Profile can testify to the control flow of the agent, and access to the Trusted Profile may also reveal the perpetrators. While this does not by any means make IID a "silver bullet" against APT, it does offer the defenders several important advantages against what has become one of the more challenging attack vectors.

1.4.3 Field Debugging

Deployment of an IID at trusted customer sites can simplify bug reporting, improve the efficiency of fault isolation, and facilitate early detection of problem scenarios. One of the central challenges addressed by research in debugging is the difficulty of correlating program errors with observable problems in program behavior. Users report their experience with a software product having no concept of the internal cause, leaving developers to search through large amounts of logically associated code for the relevant fault. Numerous techniques have been proposed to automate this process, but these tend to have high overheads, provide too much loosely associated information or rely on complex and potentially imprecise program analysis [89]. Another approach takes initiative in the debugging process by interactively leading the developer through the deduction process under the guidance of control flow and dataflow analysis [95]. The key insight is that for a developer to begin solving a problem, it is first necessary to find *where* the problem occurred in terms of program source code. An IID has an inherent advantage in this pervasive scenario because it specifically detects causes and not symptoms. When control flow diverges from the Trusted Profile, that indicates a potential cause of a problem, without any further information about what kind of problem might be caused. Coupling an IID anomaly report with an observable program error can go a long way toward preparing a developer to resolve the issue. Even if no observable error occurs (or is noticed), the cause-centric report of an IID puts a developer in the ideal position to investigate potential negative effects. Intuitively, it is easier for developers to reason forward in time than backward with respect to program execution, making the cause-centric focus of an IID report advantageous for debugging, especially where reports are coming from remote locations and are not easily replicated at the development venue.

Debugging production failures presents a similar set of challenges to field bug reports, especially where information from the failure site is limited, focuses on irrelevant information, or contains too much information [191]. Researchers have explored various techniques for improving information about remote failures, including:

- Control flow sampling to provide a snapshot of low-level program behavior leading up to the failure [127].
- Symbolic execution to replay the failing run [193].
- Search across the input domain for fault-inducing inputs [93].
- Symbolic backward execution to recover fault-inducing inputs [47].
- A control flow query engine that responds in the context of conditions known about a failure [126].

When successful, these techniques can greatly simplify the effort of diagnosing and repairing failures. But research tools are rarely used in practice, largely because they are looking for high-level information about the cause of the failure, and tools tend to provide low-level hypotheses [166]. CREDAL combines crash dumps with source code to automatically detect memory corruption leading to the crash [188]. The cause-centric logging of IID provides a similar focus on the problematic program behavior, but without speculative program analysis or dependence on potentially unavailable core dumps. Where a post-failure analysis is generally limited to identifying just one cause (at best), an IID can report multiple unexpected behaviors in a single execution, and may also have reported relevant anomalies in past executions that did not lead to a failure, or that led to a similar failure.

1.4.4 Profiling

There are also separate usage scenarios for profiling to account for differing aspects of C_{prof}:

- **Vendor profiling** relies on development infrastructure and resources to generate a valid Trusted Profile for each released module. Since updates may be frequent, the profiling procedure should be mostly automated as part of the production cycle. This is the ideal scenario for profiling but may not always be available.
- **Legacy profiling** is necessary for applications that are no longer supported by the vendor. In this scenario the monitored application is never updated, making it possible to develop a valid Trusted Profile through a more organic process. Manual effort may be required, but its value can be amortized over the lifespan of the legacy product.
- **End-user profiling** occurs for programs lacking IID support from the vendor, or legacy applications that do not have third-party support. The lack of investment implies that these will often be relatively small, simple applications that are much less challenging to profile.

Where any of these scenarios encounters challenges in sufficiently completing the Trusted Profile, it may be possible to leverage any of a variety of techniques from other research areas, including program comprehension and symbolic execution. Section 6.6 focuses on these related works in more detail.

Secure Profiling

As outlined in C_{prof}, the effectiveness of an IID in detecting malicious activity may improve with expansion of the Trusted Profile by reducing noise to better highlight important log entries, but only insofar as the added control flow edges are safe. If the adversary should gain influence over the contents of the Trusted Profile, it may be possible for future attacks to be hidden. This threat is most prominent in the end-user profiling scenario, where it may be difficult to isolate trusted inputs, or even to know for certain which inputs are trusted. But this problem can also arise in a controlled engineering environment, for example if an advanced persistent threat has established a presence in the profiling environment. These problems pervade computing in today's high-value technology, and may equally affect compiler-based CFI approaches or any other software defense. While it may not be possible to guarantee the security of profiling, several techniques can be applied to detect and eliminate adversarial influence, including validation by static analysis and delta profiling across physically isolated environments. Section 2.3 continues the discussion of this threat and potential preventative techniques in the context of all three profiling scenarios.

1.5 Prototype IID Implementations

To evaluate the feasibility of implementing an IID that meets the goals of Section 1.3, we developed two IID prototypes that are capable of monitoring some of today's most popular desktop and web applications. This research focuses on the native x86 and PHP platforms because these are widely used and feature many of the dynamic features that expose applications to attack. Although the goals for successful IID are platform independent, complications arise from today's sophisticated runtime environments that require special, platform-specific features to be implemented. For example, x86 applications frequently incorporate JIT engines that generate megabytes of code at runtime, which stretches the limits of the Trusted Profile design and ultimately requires special features to address G_{dnd}. These platform-related differences are outlined here, followed by an overview of the experimental procedures and results.

1.5.1 BlackBox for x86 COTS Binaries

Since native execution on the x86 platform can only be monitored by hardware counters that are sparse and/or coarse-grained and have limited scope of introspection, BlackBox is implemented as a client of the binary translation framework DynamoRIO [14]. An additional benefit is that BlackBox is compatible with commercial off-the-shelf (COTS) binaries, avoiding dependence on program source code. But it requires adding a security API to the DynamoRIO core, because BlackBox needs information about low-level control flow that is not typically necessary in binary translation use cases. The implementation of IID in BlackBox is mostly platform agnostic, with three main exceptions:

- The architecture of today's best-performing binary translators is optimized for statically compiled code but performs very poorly for dynamically generated code (DGC), such as a browser's JIT engine produces when executing hot JavaScript paths in web applications. A special DGC optimization is required for BlackBox to meet the expectations of G_{perf}.

- The prevalence of DGC in x86 applications also raises complications for the Trusted Profile, because random factors within popular JIT compilers cause arbitrary differences in the monitored execution traces, resulting in a storm of spurious anomaly reports and failure to meet G_{dnd}. BlackBox resolves this by making a special case in the Trusted Profile for DGC.

- Since the semantics of x86 machine code ignore function boundaries, exploits are able to link together small fragments of code to compose malicious behaviors. This makes it necessary to report intra-procedural anomalies, but benign instances of these anomalies are frequent in today's complex programs, compromising G_{dnd}. BlackBox accounts for the noise of benign anomalies by inferring a degree of suspicion based on program execution history.

The BlackBox prototype currently supports 32-bit Windows 7 applications, a platform we chose for its wealth of compatible malware, though implementations for other Windows or Linux environments would be similar. In controlled experiments, BlackBox logged fewer than 100 anomalies per hour—all ranked at low suspicion—during normal usage of a broad range of popular Windows applications including Microsoft Office and Google Chrome. Although sophisticated exploits are rarely available for these applications, because vendors pay large bounties to prevent distribution, BlackBox detected 100% of known exploits against smaller applications like Notepad++. The runtime overhead of BlackBox on the SPEC CPU 2006 benchmarks is 14.7%, and independent third-party reviewers confirm that BlackBox causes only minor degradation in user experience for popular and performance-intensive Windows applications.

1.5.2 ZenIDS for PHP Applications

Since the PHP interpreter directly loads application source files for execution, control flow introspection can be instrumented in the core interpreter. ZenIDS is implemented as a dynamically linked PHP extension that requires 4 additions to the interpreter's extension API, along with 2 hooks each in the standard OPCache and MySQL extensions. A platform agnostic implementation of IID is functional for PHP applications as sophisticated as WordPress, though there are again three exceptions that require special features for ZenIDS to approach the IID goals:

- To monitor the intricate semantics of PHP script that arise from dynamic programming constructs, such as importing a script through a string variable, ZenIDS reduces PHP control flow to a significantly simpler abstraction.
- Although the HTTP convention of dividing user input into discrete requests tends to simplify monitoring and G_{dnd}, other complications arise from a dynamic characteristic of the PHP development ecosystem. PHP applications blur the line between development and administration by providing large sets of optional features that can easily be activated through the user interface. Feature changes that would require a newly compiled executable on the x86 platform can occur in PHP with the simple click of a radio button. To avoid regenerating the Trusted Profile every time this occurs, ZenIDS implements an *evolution* feature that recognizes site changes made by trusted users and dynamically expands the Trusted Profile accordingly. Even if the vendor provides a Trusted Profile for each PHP module, there are thousands of WordPress plugins and the average site has more than 20 installed [55], making the evolution feature necessary to cover this vast domain of potential cross-module edges.
- A naïve implementation of ZenIDS has $10\times$ overhead vs. an optimized LAMP stack. The prototype implementation is able to address G_{perf} with conventional optimizations.

The ZenIDS prototype extends the 64-bit x86 Linux build of PHP 7.0 alpha 3, which was the latest available at the time of development. In a 360-day experiment monitoring WordPress and two other PHP applications, the Trusted Profile was trained exclusively by dynamic profiling in as few as 298 HTTP requests and logged just 7 spurious anomalies while correctly identifying over 35,000 malicious requests. ZenIDS also detected 100% of known attacks during monitoring of a WordPress instance that was configured with vulnerabilities from the last 10 exploits published on `exploit-db.com`. After optimizing the ZenIDS extension and interpreter instrumentation, runtime overhead for a random slice of traffic from the 360-day experiment is under 5% for all three applications.

1.6 Contributions

This dissertation makes the following research contributions:

- A software monitoring system called Introspective Intrusion Detection that combines the precision of CFI with the broad scope of anomaly detection, making it sensitive to unforeseen RCE takeover methods as well as the payload while logging a low rate of spurious anomalies.
  - A logging system that greatly simplifies forensic analysis of security incidents and debugging of security vulnerabilities by selectively recording detailed control flow events and related context whenever suspicious behavior is detected.
  - A system of modular trust that automatically incorporates all libraries used by a program, limits branch targets across modules to observed usage, and can be extended with whole-program context sensitivity at every indirect branch, including DGC and OS callbacks.
- Prototype implementations of IID that detect known attacks along with exploits occurring in the wild while maintaining user expectations for the runtime performance of today's most popular applications, including Google Chrome, Microsoft Office and WordPress.
  - BlackBox[2], an implementation of IID for 32-bit x86 COTS binaries on Windows 7.
  - ZenIDS[3], an implementation of IID for PHP 7 on 64-bit Linux.
- A profiling technique for capturing an abstraction of application control flow into a Trusted Profile that can accurately distinguish normal (safe) control flow events from anomalies having a high probability of malicious intent.
- An evolution technique for expanding the Trusted Profile when application structure is modified at runtime by authenticated users having trusted accounts.

[2] Available at `https://github.com/uci-plrg/blackbox`.
[3] Available at `https://github.com/uci-plrg/zen-ids`.

- A permission model for detecting anomalies in dynamic code generators that is resilient to transient differences in the structure of the dynamically generated code.
- An optimization technique for binary translation of dynamically generated code[4] that reduces overhead to under 3× on industrial JavaScript benchmarks.

1.7 Organization

The remainder of this dissertation focuses on the implementation of an IID, including technical presentation of our two prototypes, and our experimental evaluation of the prototypes on the basis of the IID goals outlined in Section 1.3. To give an overview of its internal structure, Chapter 2 presents the generic features of an IID based on a hypothetical platform where control flow introspection is trivial to obtain and there are no challenges raised by performance characteristics or prevalent software architecture. Chapter 3 presents the implementation details of BlackBox, focusing on platform-specific features that are necessary to address the IID goals, and concluding with an experimental evaluation. BlackBox requires an optimization of DynamoRIO for DGC to meet goal G_{lite}, which is presented separately in Chapter 4 because it is orthogonal to the concerns of IID and security in general (for example having use cases at Google and ARM focusing on software engineering). Chapter 5 parallels Chapter 3 in presenting the technical details and experimental evaluation of ZenIDS. To complete the motivation of IID and establish its position in the security research landscape, Chapter 6 expands on Section 1.1 with a more detailed discussion of representative works, focusing on CFI and traditional intrusion detection, along with a taxonomy of RCE exploits and the issues related to detecting them with an IID. Chapter 7 concludes with a discussion of potential next steps towards industry adoption of Introspective Intrusion Detection.

[4] Available in the DynamoRIO GitHub repository under branch `experimental-optimize-jit`.

Chapter 2

Paradise: a Hypothetically Pure IID

Figure 2.1 presents an overview of Paradise: **P**latform **A**gnostic **R**CE **A**ttack **D**etector for an **I**deal **S**ecurity **E**nvironment. This IID is designed for a hypothetical platform where all the advantages of IID are in effect, but none of the challenges are present (as enumerated in Section 1.2). The purpose is to isolate the basic usage model and internal structure of an IID from the torturously twisted features that are often necessary for a real implementation to be effective in practice.

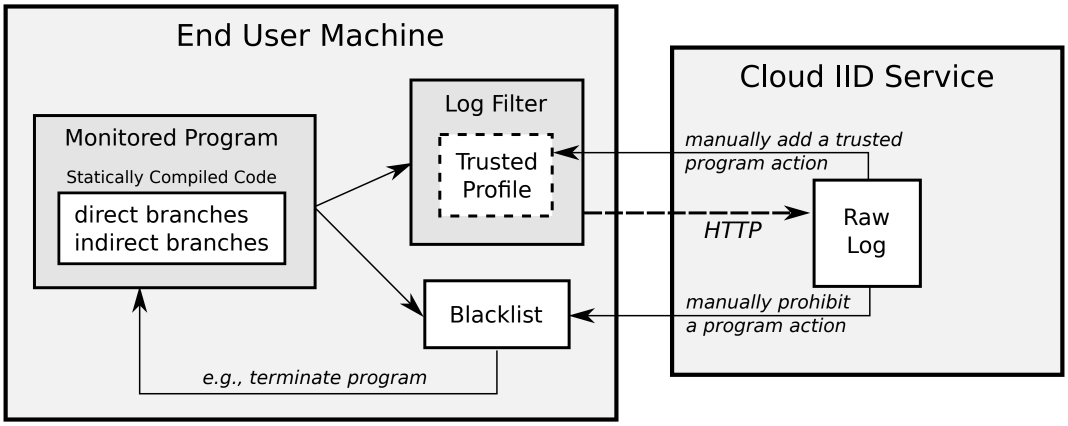


Figure 2.1: Component connectivity overview of application monitoring in Paradise.

Section 2.1 enumerates the *platform snares* that have been hypothetically banished from the land of Paradise. The adversary, however, does have permission to roam here, and the platform-agnostic threat model is presented in Section 2.2. Then, from this last glimpse of reality, we descend into the idealized incarnations of the IID deployment and maintenance phases in Sections 2.3 and 2.4.

Cloud IID Service

Throughout this dissertation we assume the availability of a cloud-based service provider that receives connections from end-user IID instances to securely store Trusted Profiles and log streams. The service can be extended to provide analysis for various purposes such as suspicion ranking of log entries, or inference of suspicious trends among groups of users. When vendors deploy IID at customer sites to facilitate forensic analysis and debugging, they can also provide the IID Service for those instances. Most of the service components have been implemented as individual shell programs for use in the prototype experiments, and the corresponding github projects are linked from the README files of the IID prototype repositories. Secure storage and streaming were omitted from the IID prototype experiments because they represent security challenges that are already well understood.

2.1 Platform Snares

The central question addressed in this section is: why do we need a separate design and implementation of IID for each platform? In any project there may always be a few platform-specific code and integration differences, for example the C code for the PHP MySQL extension requires OS-specific `#ifdef` clauses, even though at runtime it integrates with PHP in terms of identical API calls and data structures on any platform (and the IID prototype for PHP is little more than a PHP extension). But there are major design differences between the IID prototype for x86 binaries and the one for PHP. These differences account for the *platform snares* presented in this section, which range from subtle matters of programming convention to major infrastructural components like script interpreters and process virtualization runtimes. If it were not for the snares, a single IID design would be sufficient for all platforms, and there would only be a few low-level differences between the implementations.

This raises a second pivotal question: for the purposes of IID, what exactly constitutes a platform? Today's software is commonly built as a composition of two or more layers of abstraction, mainly to improve development efficiency, such that the semantics executing at one layer are invisible to an observer of another layer's control flow. For example, consider a trivial Turing machine that is implemented in less than 100 lines of code, yet is capable of running infinitely complex programs. The execution traces for two such programs may be very different when the traces are recorded in terms of Turing instructions—but if the traces are recorded in terms of the Turing machine implementation, they will often be identical (depending on context and flow-sensitivity). In this sense, the Turing script is *opaque* from the perspective of the native implementation of the interpreter. Accordingly, for the purposes of IID, the Turing machine is a separate platform from its native host platform. An IID implementation can only monitor programs for the one platform it is designed for, so if there are multiple platforms in a process—such as the common case of a browser rendering JavaScript—then two separate IID instances are required to monitor the entire process.

This factor leads somewhat circularly to the first platform snare. If an interpreter lowers part of the interpreted script by dynamically generating code for the host platform—usually native machine code, as in a JIT compiler—the lower-level IID will observe control flow from two separate platforms. This problem does not apply to DGC in general, but arises in the interface between two independent platforms. For example, the entry points from interpreted code to JIT generated code are difficult for the lower-level IID to profile: since the behavior of the interpreted code is opaque, all the entry points will be aliased to a single source node in the Trusted Profile of the interpreter. This weakness is exploitable and requires special handling in the IID, as described in Section 3.4.

Snare $S_{\textbf{opq}}$: semantics become **op**a**q**ue across layers of abstraction

Today's software is built as a composition of two or more layers of abstraction such that the semantics executing at one layer are opaque to an observer of another layer's control flow.

An IID records and monitors control flow in the form of a model, usually a control flow graph, that represents the actions of the program as it executes. The set of possible actions is specified by the runtime platform, and may include many details that are not important for security purposes. For example in the ISA of a typical CPU, most instructions simply fall through to the physically subsequent instruction, so it is not useful for a control flow monitor to include every machine instruction in its execution traces. Similarly, interpreted languages often define a complex set of control flow edges such as callbacks from the interpreter to user functions, invocation of code defined outside of any function, and implicit invocation of one or more functions when various pre-defined conditions are met. For security purposes, it can be much more efficient to model all of these edges with a single edge type, without loss of accuracy. But these details are not permitted in the land of Paradise because they are not fundamental to the construction of an IID.

Snare $S_{\textbf{cf}}$: verbose or intricate **c**ontrol **f**low

Verbose or intricate control flow can encumber a literal execution trace with details that are not relevant for security purposes, and may cause false positives.

Some platforms allow execution to enter a procedure at any location in the procedure body, without ever using the entry point. While this can be leveraged for advanced optimization techniques, it also exposes a wealth of opportunities for attackers to manipulate control flow. Such platforms typically do not constrain the `return` instruction to the call stack, allowing an exploit to control execution by fabricating return addresses. Even worse, some platforms optimize for code size by omitting instruction boundaries, which inadvertently allows execution to begin in the middle of a multi-byte machine instruction (thereby changing the semantics of the instruction).

Snare $S_{\textbf{sem}}$: lack of conventional **sem**antic boundaries

The runtime environment may not enforce call/return semantics or the boundaries of functions or instructions.

The concept of the Trusted Profile is only applicable to programs that load the same code on every execution. But many platforms allow a program to dynamically load modules, or to generate new code as it runs. Program modules may also change, for example during software updates. An IID may need to modify the Trusted Profile accordingly, or represent fluctuating code regions with an opaque abstraction.

Snare $S_{\textbf{cc}}$: **c**ode **c**hanges

Code changes may occur at any time.

Since code changes are often complicated and prone to failure, some programs take a modular approach that can be equally difficult for the Trusted Profile approach. These programs incorporate vast frameworks of generic functionality and provide the user with a configuration interface that can quickly enable or disable major application features. For the Trusted Profile approach, it is as if the newly enabled features were loaded at runtime, since trust has never been established for the code implementing those features. Third-party plugins have a similar effect, along with code customization on scripting platforms.

Snare $S_{\textbf{late}}$: **late** feature binding

New features may be integrated after deployment, whether by activation from frameworks, installation of plugins or code customizations.

Many of the IID advantages and challenges pivot around user interaction, for example G_{dbg} requires precise logs that are also understandable for the purposes of forensic analysis and program debugging. But an IID implementation can only monitor the application in terms of its executable code, which may have undergone complex transformations during the development process that obfuscate the high-level semantics. For example, it can be very difficult for even the most sophisticated analysis to determine which application behavior(s) are performed by a given segment of compiled x86 instructions. This obfuscation can interfere with the effectiveness of IID in software development and security analysis use cases.

> Snare $S_{\textbf{obf}}$: **obf**uscated code
>
> Complex transformations such as compilation may obfuscate the source-level semantics of application control flow.

It is not easy to foresee all the possible complications that might arise in other runtime platforms, especially those that are not common today. For that matter, at the beginning of this project we did not anticipate several of the issues that did arise in our IID prototypes, despite having substantial experience with those platforms. So this discussion of snares should be considered a representative sample of potential issues that can arise in real deployment environments.

2.2 Generic Threat Model

Although the land of Paradise is free of platform snares, it is by no means free of security snares—the adversary has non-privileged access to the monitored program and is able to craft input that causes unexpected program behavior. Access to the filesystem is limited to privileged users and the application, so the adversary can only create or modify files by using (or abusing) application functionality. If the adversary gains control of an application on the monitored machine, it is assumed the exploit may escalate to root access and take control of the machine. Security breaches in the IID Service are either impossible or effectively mitigated by administrators, such that the service is always available to securely store log streams and provides correct Trusted Profiles. To obfuscate side channels related to the IID Service, the Paradise runtime randomly sends empty anomaly reports when necessary to maintain consistency in the rate of communication. The source code for Paradise and the monitored application are publicly available, and the adversary can determine which version is running. The time and duration of profiling is only known to privileged users of the application and the IID Service.

The adversary may mount an advanced persistent threat against the organization that administers the IID. Malicious components within the infrastructure may influence the execution of any programs, but orthogonal security measures prevent such components from directly modifying files or communicating with entities outside the organization. Invasive components may include resident malware and malicious circuits in commodity hardware. However, no APT is omnipresent—for any physical location where the adversary has influence, there is some other location, perhaps at a third party site, where the adversary does not have influence.

2.3 Deployment

Several steps are involved in preparing and using an application with an IID. The runtime can be deployed to the end user machines using a standard installer that is configured to connect the IID to the vendor's IID Service. Since the primary use cases for IID are software development and security analysis, it is assumed that the vendor or analyst is responsible for generating and maintaining the Trusted Profile. This may still involve profiling end users, for example in the case of monitoring legacy software for which there are no automated tests or profile guided optimization (PGO) samples. In this case the IID Service host will coordinate profiling and compilation into the Trusted Profile.

2.3.1 Generating the Trusted Profile

In profiling mode, an IID observes program executions and generates traces representing the security-relevant aspects of the control flow. In the absence of platform snares, the Trusted Profile can be defined as a trivial call graph, where a node represents a function or a system call number, and an edge represents a function call or a system call (Figure 2.2). It is sufficient to only monitor the call graph because the adversary relies on system calls to carry out malicious actions—manipulating control flow within a function can never directly harm the system. An offline tool merges these execution traces into a composite call graph, which becomes the Trusted Profile.

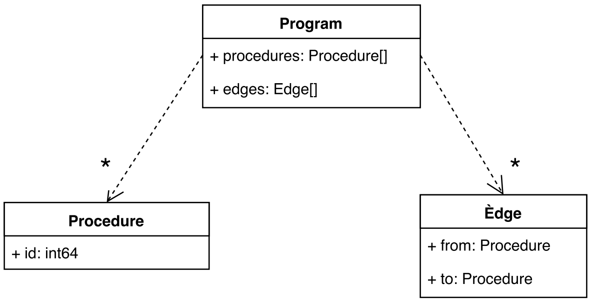


Figure 2.2: The Trusted Profile as defined in Paradise.

The two prototype implementations of IID demonstrate effective results for G_{wx} and G_{dnd} without including context or path information in the Trusted Profile—it is simply a bag of nodes and edges. This is because, to the best of our knowledge, real-world adversaries are so far not able to construct an entire exploit payload using exclusively the set of edges normally executed by the application. If it becomes necessary, the Trusted Profile can be extended with some degree of context and flow information, though at significant performance cost (as discussed in the introduction of the IID advantage A_{wpm}).

The profiling phase is easy in Paradise where application vendors always have automated tests that cover every intended control flow path. In the case that users have installed third party plugins or other extensions, it may be necessary to profile the control flow edges between the application and its optional components. This can be done at the vendor's site, because in Paradise the automated tests invoke all possible callbacks from dynamically integrated components. To protect the Trusted Profile from tampering (in the case of a successful exploit), the IID Service makes profiles exclusively available for download through an authenticated connection. Although the adversary may be able to obtain and analyze the Trusted Profile, a separate attack on the cloud service would be required to tamper with stored profiles.

Advanced Persistent Threats

A high-level perspective on the potential role of IID in APT defense is presented in Section 1.4.2. This section focuses on low-level vulnerabilities of an IID that may become a target during an APT attack, and the technical facilities that can protect the IID.

The vendor's profiling site is not necessarily secure from all adversarial influence. A premeditated attack may deploy low-impact agents to the vendor's site through malware or perhaps social engineering. For example, a recent study of more than 200,000 Android applications, 97.9% contained at least one insecure code snippet that had been evidently copied from Stack Overflow or a similar discussion platform [56], suggesting that a crafty adversary may be able to pollute any part of a vendor's codebase through seemingly innocuous channels. It may also be possible for untrusted commodity hardware to contain malicious circuits under the attacker's control. This kind of threat can affect the vendor's toolchain at all levels, for example by manipulating intermediate files during compilation to disable CFI constraints. But the IID profiling process is likely to be one of the easier targets, since any alteration of control flow in the profiled application will simply be recorded and trusted.

Potential counter-measures will be presented here hypothetically, and concrete experiments are reserved for future work. Since the adversary is not omnipresent, one approach to detecting an advanced persistent threat focuses on the differences in execution between physically isolated environments. This can be as simple as manually examining the control flow delta between profiling sessions at different sites. If sufficient care is taken to transfer the application to both sites without carrying along potentially infected elements from the development environment, a deterministic profiling procedure should reveal any unexpected discrepancies. Since these may be benign, for example caused by configuration or network differences, the audit process may be labor intensive. However, in a high-value organization, this kind of precautionary measure would be considered routine. Differencing techniques have been successfully applied to anomaly detection, where pollution of trained models can be automatically detected in the delta between independent sites, though the automation technique may not be compatible with the large footprint of the Trusted Profile [35].

In anomaly detection for financial applications, a similar technique correlates results from randomized sub-models to accurately isolate outliers in datasets [102]. For regions of the Trusted Profile that remain under suspicion after cross-site differencing, subset diversification of the Trusted Profile may help experts evaluate these program behaviors. After marking these edges as suspicious, the IID runtime can periodically omit suspicious edges from the Trusted Profile to sample usage of the corresponding control flow that occurs in the deployment environment. After sufficient time for collection and analysis, experts can determine whether to trust the questionable edges, or to blacklist them while investigating potential causes.

A simpler approach would evaluate the Trusted Profile according to a conventional static analysis. The effect would be similar to πCFI [125], which dynamically enables groups of control flow edges when the entry point is encountered, but falls back on a statically derived CFG to limit adversarial activation. Though since a static analysis can be more permissive than the strictly dynamic Trusted Profile, this approach would not detect adversarial influence that conforms to the static CFG.

The risk of Trusted Profile pollution for end-user profiling scenarios is more difficult to mitigate, though similar techniques may be effective. Trusted communities of users are also explored in [35] and may lend a useful dimension to the proposed differencing analysis. But in general, low-investment application deployments tend to be highly insecure and replete with orthogonal opportunities for compromise, limiting the value of a dedicated effort to tightly secure IID profiling for this scenario. In our experience with profiling a live PHP website, the Trusted Profile was highly polluted with typical malicious requests (which comprise 25% of traffic on our research lab homepage), yet the IID remained substantially effective (Section 5.5).

2.3.2 Monitoring

The system administrator installs the IID runtime onto the machine where the protected applications will be executed, and configures it to access the Trusted Profile. For distributed applications, a separate IID would be installed on each node. The installation procedure binds the protected programs to the IID, such that they are always launched in the IID runtime. By default the Trusted Profile is stored by the IID Service, and local copies may be maintained under regular MD5 verification.

Monitoring focuses on instruction execution and also module loads, which in Paradise only occur at program startup. Paradise compares each loaded module to a signed copy of the corresponding binary that was loaded during profiling. If there is any difference, a module mismatch anomaly is reported. Then as the program executes in the IID runtime, control flow is continuously compared against the Trusted Profile. If the execution takes any control flow edge that does not appear in the Trusted Profile, the IID logs an anomaly indicating the untrusted edge. Similarly, if the execution takes a control flow edge that matches a blacklist entry (Section 2.4.1), the IID takes the blacklist action (for example, terminating execution).

In its default configuration, the IID generates one simple log entry for each anomaly, indicating the untrusted control flow edge, a timestamp, and the process where it occurred. Additional information can be included to assist in forensic analysis and debugging, such as:

- a stack trace
- the thread hierarchy
- the process hierarchy
- a system call trace
- a complete control flow trace
- a complete trace of the entire process tree

The log content for an anomaly can be configured with a predicate, for example to log more information for anomalies occurring in a particular module.

Logging Service

Although various configurations are possible, by default Paradise sends logs entries in realtime to the IID Service over a write-only HTTP socket. This prevents the adversary from tampering with the logs after a successful exploit. It also simplifies visualization tools, which can be centralized in the IID Service and provided through a web interface or a thin client on the IID host machine. The service is capable of throttling a connection if an attacker gains control and attempts a DOS attack. To prevent the adversary from leaking information through side channels related to the IID Service, the runtime sends empty log entries at random intervals, which increase in frequency as the rate of real log entries declines, and vice versa. The entries are encrypted to prevent the adversary from discovering which entries are empty.

While the focus of IID is on forensic analysis and vulnerability debugging, it can also be an effective tool for preventative analysis that can discover potential problems early. For example, the IID Service can correlate similar anomalies reported across machines and recommend investigation when suspicious behaviors appear to be spreading, or start occurring globally without precedent. For example:

- Suppose a new game becomes popular, and it installs a customized version of a shared library in an unconventional way. If monitored applications accidentally load the customized version of the library, anomalies will be reported even if the application appears to work correctly. This kind of problem can be extremely difficult to diagnose on the basis of symptoms in the adversely affected applications, but is trivial to understand from module mismatch reports.
- A study of malware development tools found in the wild [86, 85, 159] shows substantial dependence on trial and error, especially during the deployment phase. Initial versions of the exploits worked locally but failed on target machines, leading to a cycle of minor revision and redeployment. Conversations on “underground” malware forums indicate that reliability statistics are a major selling point for malware. While the reputation protocol in these communities is somewhat opaque, it appears that new malware is deployed on various test targets to “prove its worth” for marketing. The IID Service can detect these early deployments and immunize all customers with corresponding blacklist entries before a major malware attack is attempted.

Log Usability

In Paradise the development tools are so well structured and reliable that many software engineers are unfamiliar with the mechanics of low-level control flow, or how the source code is lowered to that form. Since the IID logs can only report anomalies in terms of executable elements such as basic blocks, it can be difficult for a developer in Paradise to understand a reported attack and diagnose the compromised program vulnerability. Conversely, security analysts are mainly familiar with executable forms and are not easily able to communicate about security incidents in terms of program architecture or user-facing components. Fortunately, the program analysis tools in Paradise are as effective as the development tools, making it possible to extract the correlation between compiled instructions and source code, or between compiled functions and application features. In the real world these challenges have also been addressed in research [144, 178], and are further discussed in Section 6.6.1.

2.4 Maintaining the Trusted Profile

Although the Trusted Profile here in Paradise contains every intended execution path in the monitored program, not all IID deployments necessarily benefit from total coverage. The boundary between malicious input and ordinary user error can be subtle, making it potentially valuable to log anomalies for program paths such as error handling that were intended by the program authors to be used. For example, suppose a program uses a third-party authentication mechanism such as Google Sign-In, and the developers are not confident that they understand all the corner cases. To harvest use cases from deployed instances of the program, the developers can remove those regions from the Trusted Profile. Conversely, developers may discover an error in their code through IID anomaly reports, and find that the frequency of occurrence becomes a nuisance for routine log analysis. If it may take some time to deploy a patch to the monitored sites, the developers can in the meantime whitelist the unintended control flow by manually adding it to the Trusted Profile.

2.4.1 Blacklisting

Security vulnerabilities are more easily discovered than fixed, causing an inevitable delay in the release of patches to prevent known exploits. Software development and security organizations may have clients who can benefit from an intermediate resolution. The IID blacklist can facilitate this service by taking any configurable action when prohibited control flow is observed. The predicate can be as simple as an control flow edge specification, just as it appears in the IID log, or it can take into account any of the dynamic factors that are visible in the IID runtime.

In the case of end-user administration of an IID, blacklist entries can either be taken directly from the IID log, or from trusted members of a security community. This kind of safety measure is common in the PHP ecosystem today, where trusted experts post nominal patches for known vulnerabilities, and website administrators of all levels of expertise download and integrate them into their site's PHP source code. Distribution of blacklist entries can be done in much the same way, perhaps with greater reliability considering that the transferred artifact is much simpler than a raw code fragment.

Chapter 3

BlackBox

The difference between Paradise and a real IID for x86 commercial off-the-shelf (COTS) binaries lies simply in the four IID challenges (Section 1.2) along with the following four snares (Section 2.1) introduced by the x86 platform:

$S_{\textbf{sem}}$ Several aspects of the source program's semantics are omitted from the x86 ISA:

- The `ret` instructions obtain the return address from a stack slot, making it possible to break the symmetry of any call/return pair by returning to an arbitrary address.
- Procedure boundaries are only encoded in debug symbols, and since these are never referenced by x86 branch instructions, it is possible to enter a procedure at any byte.
- Instruction boundaries are not encoded anywhere in the binary executable image, but the ISA is dense enough that in many cases, entering an instruction in the middle will result in execution of a valid but different instruction.

These factors concede a wealth of opportunities for the adversary to fabricate control flow paths that were never intended by the program's developers. While some of these issues are addressed by the Intel "Control-Flow Enforcement Technology" (CET) feature, Section 3.1

shows that even for processors having CET available and enabled, the effects of S_{sem} do not necessarily change. Given the ongoing possibility of these unexpected behaviors, an IID should be able to continuously monitor return targets along with all intra-procedural control flow. The addition of these low-level details makes an IID more likely to exhibit small variations between observationally equivalent executions, increasing the rate of spurious anomalies.

$S_{\mathbf{cf}}$ Most x86 instructions do not have any direct effect on control flow—simply falling through to the physically subsequent instruction—making them irrelevant to control flow security. Another complication occurs in callback mechanisms:

- In Windows, many system call implementations make a callback to a user-space function, which can confuse the shadow stack, leading to artificial context sensitivity for nested chains of alternating system call and callback.
- Callbacks are also a popular idiom within Windows applications, but the physical control flow specifies the callback target as an address, which does not respect modularity. For example, if the Trusted Profile was generated separately for two modules having a callback edge between them, an explicit address for the callee will often be incorrect, even if it is logically the same callback.

Conversely, the Windows API establishes a convention of wrapper functions for system calls, making it more difficult to detect whether untrusted control flow may affect system calls.

$S_{\mathbf{cc}}$ Prevalent operating systems for x86 allow code changes to occur at runtime:

- Modules can be loaded and unloaded via system call.
- Code within a loaded module can be overwritten and executed.
- Code can be written to dynamically allocated memory and executed.

$S_{\textbf{opq}}$ The software development ecosystem for x86 platforms relies heavily on layers of abstraction, both in software construction and in runtime systems, making it more difficult for an IID to generate log entries that the user can easily understand and manage. For example:

- Popular compilers transform source code into assembly language, and transform again into machine code, making it difficult to correlate elements of the resulting binary image with the components and behaviors of the original source program.
- It is common for x86 programs to embed script interpreters whose semantics are opaque to an IID that monitors native execution.
- Many of these interpreters incorporate JIT compilers (Table 3.1), which may require an IID to substitute corresponding portions of the execution trace with an abstraction.

For example, Adobe PDF Reader executes native x86 code generated by several compilers:

- The Visual Studio C/C++ compiler.
- The .Net Native compiler for C# and/or Visual Basic.
- The Microsoft Managed Runtime JIT that generates utility routines to transform data structures between the managed and unmanaged formats.
- The Adobe Flash JIT (`npswf32.dll`).
- The Internet Explorer JIT (`jscript9.dll`).

	Dynamic Routine Generators	JIT Compilers
Word	8	1
PowerPoint	9	1
Excel	4	1
Outlook	4	1
Chrome	6	2
Adobe PDF	13	2

Table 3.1: Number of distinct modules employing code generators in popular Windows programs. Dynamically generated code is a growing trend.

Figure 3.1 depicts an overview of the components that BLACKBOX adds to Paradise so that it can survive the IID challenges and the snares of the x86 platform. After Section 3.2 makes an adjustment to the threat model, Section 3.3 presents the BLACKBOX runtime and its format for the Trusted Profile, which accounts for S_{cf} and a special case of S_{cc} where code in statically compiled modules is overwritten at runtime. The section concludes with the **Shadow Stack**, which accounts for the S_{sem} case of vulnerable `ret` instructions. In Section 3.4, three new runtime events for **Dynamically Generated Code** negotiate layers of abstraction created by script interpreters and JIT compilers, and three corresponding edge types are added to the Trusted Profile. The vulnerabilities exposed by S_{sem} require BLACKBOX to monitor programs at a granularity too high for G_{dnd}, yielding a significant rate of benign anomaly reports. Section 3.5 presents the **Stack Spy** and **Sniffer Dog** which can combine to distinguish important log entries from safe irregularities in low-level control flow. To demonstrate the potential of BLACKBOX to perform as a real-world IID, Sections 3.5 and 3.6 present case studies and experiments that evaluate the BLACKBOX prototype in terms of G_{dnd}, G_{kx}, G_{rx}, G_{blk} and G_{dbg}. Section 3.6.3 discusses the development and maintenance cost of BLACKBOX and its supporting IID Service components.

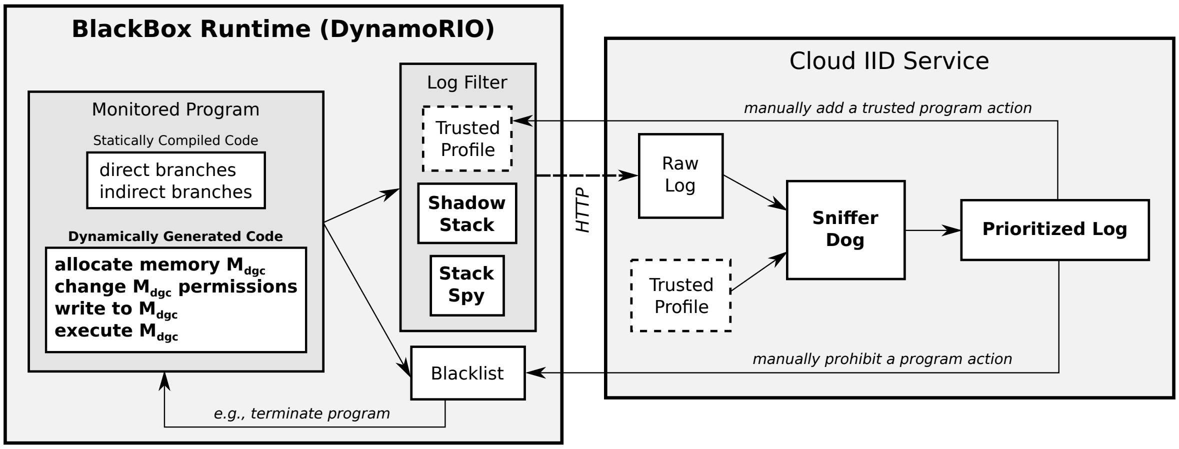


Figure 3.1: Component connectivity overview of application monitoring in BLACKBOX. Bold font indicates components that were **not necessary in Paradise** (Chapter 2), but are necessary on Windows for x86 processors to account for the four IID challenges (Section 1.2) and four snares arising from the x86 ISA and the Windows software ecosystem (Section 2.1).

3.1 Intel CET

In 2016, Intel released preview documentation for a new featured called “Control-Flow Enforcement Technology”, which extends the x86 ISA with instructions and a shadow stack to enforce both forward and backward CFI [82]. The feature has not been implemented in any released Intel CPU up to today (August 15, 2017), as evidenced by the fact that all Kaby Lake processor models clearly indicate support for SGX and likewise indicate *no support* for CET [84]. Subsequent processor models Coffee Lake and Cannonlake have not been released yet [83]. Furthermore, even though the new CET instructions are already decoded to NOP on all Intel processors since Haswell (for backward compatibility, see Section 1.2 of the CET Preview), Microsoft has yet to implement support for CET in the Visual Studio compiler or any of its major product lines, including Windows and Office [111] (other supporting facts restricted by NDA).

Since version 2.0 of the CET Preview was just released in June of 2017, it is difficult to assess the impact of the feature on S_{sem}. But if the current preview holds, it will not necessarily allow for any simplifications in the design of BLACKBOX, even when the majority of x86 users have access to both software and hardware that enable CET protection:

- The forward branch integrity technique defines a single class of indirect branch targets, such that any target is valid for every indirect branch source in the application. While this precludes instruction splitting and eliminates a large percentage of other potential gadgets, it does not constitute enforcement of function boundaries. For example, an indirect call can still target a case label in the middle of another function without raising a CET hardware exception.
- The CET shadow stack is incompatible with current implementations of Microsoft Office and Adobe PDF Reader, which implement soft context switching in hand-coded assembly, whereas CET requires every soft context switch to use the Interrupt Stack Table (IST). These assembly fragments were implemented in Office more than two decades ago (Derek Bruening, personal communication). Similar constructs in `ipsecproc.dll` make a soft context switch into a task that is implemented in dynamically generated code, making it fundamentally incompatible with the IST and therefore incompatible with CET—yet this operation executes during startup of every application in the Microsoft Office 2013 suite.

Although the presence of a hardware shadow stack may alleviate performance-intensive checks in BLACKBOX for some applications, it would not make a significant improvement in overall performance, since the majority of overhead comes from indirect branches translated from the Windows IAT. With these points in mind, the remainder of this chapter does not make further reference to the pending CET release. Section 6.2 presents the potential future scenario where CET is ubiquitous on the x86 platform and all major applications have been updated to support it, discussing the changes that could make to the security landscape, and the consequences for the necessity and effectiveness of Introspective Intrusion Detection.

3.2 Threat Model

The threat model for BLACKBOX extends the threat model from Paradise (Section 2.2).

- The adversary is potentially able to modify the target of any branch to reach any location in memory (in some cases limited in range by the semantics of individual instructions).
- The adversary may be able to gain arbitrary read/write access to memory within the confines of the Trusted Profile through a Control Flow Bending attack, but is not able to locate the BLACKBOX data structures in memory without causing untrusted control flow to occur in exception handlers.
- A targeted attack against BLACKBOX may be able to disable it, but if the attack occurs while BLACKBOX is monitoring and the Trusted Profile is sufficient, the attack on BLACKBOX will be recorded to the IID Service before BLACKBOX is disabled.

If advances in attacks like CFB make this threat model obsolete, BLACKBOX can potentially protect itself by adding context sensitivity (see A_{wpm}), though the technical details of those potential improvements are reserved for future work. Alternatively, a recent technique in distributed dataflow analysis is able to detect known vulnerabilities at just 1% overhead on commodity hardware, and may be effective against non-control data attacks such as CFB [71]. The assumption throughout the remainder of this dissertation is that non-control data attacks are defeated by means that are not presented here.

3.3 Monitoring

Since program introspection is not natively supported on x86 processors, the BLACKBOX runtime is implemented as a DynamoRIO client that requires a security API added to the core framework.

Under the BlackBox runtime, the operating system loads the program modules into memory in the normal way, but the code is no longer executed directly from those mapped images. Instead, BlackBox copies program instructions into a code cache as the execution encounters them, and execution occurs over the copy. Since BlackBox has exclusive control over the contents of the code cache, the observation of a control-flow branch when it is initially linked within the code cache remains valid for the duration of the execution in the majority of cases. This makes it possible for BlackBox to observe every branch while achieving near-native performance (see Figure 3.5).

3.3.1 Trusted Profile

The simple call graph used in Paradise for the Trusted Profile is not sufficient for monitoring the loose semantics of the x86 ISA. Figure 3.2 depicts the node and edge types represented in the Trusted Profile used by BlackBox. The node and edge types for dynamically generated code, labeled `DGC*` and `gencode_*`, will be presented later in Section 3.4.

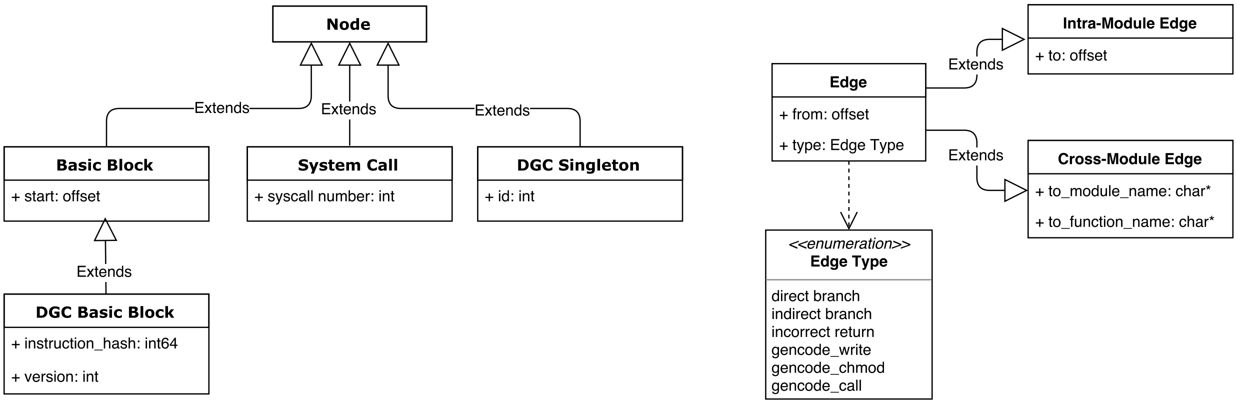


Figure 3.2: BlackBox Trusted Profile (simplified). Since dynamically generated modules are anonymous, BlackBox assigns arbitrary names and identifies them by instruction content.

Since execution can enter a procedure at any location using control flow instructions such as `jmp`, `call` and `ret`, any control flow branch is a potential security risk. This makes it necessary for BlackBox to include all control flow branches in the Trusted Profile. But the majority of instructions in the x86 ISA do not have any effect control flow, simply falling through to the next instruction, which makes it impractical to model every instruction as a Trusted Profile node. Instead, the Trusted Profile focuses at the granularity of the basic block, defined as a sequence of instructions having one entry at the top and one or two exits at the bottom.

Various programming idioms in the x86 ecosystem use the `ret` instructions in an unconventional way by writing an address into the return slot on the stack before a `ret` is executed. While in many cases this corresponds to a soft-context switch, which BLACKBOX can account for dynamically (see Section 3.3.2), it is also used in ways that are less conducive to systematic recognition. Instead of trying to understand the intention behind these unusual `ret` instructions, BLACKBOX simply adds a Trusted Profile edge of type *incorrect return* when necessary.

3.3.2 Trusting Returns

The natural way for an IID to distinguish incorrect returns is to assign categorical trust to normal returns. The intuition is based on the symmetry of call/return semantics: if the call was trusted (or has already been logged), the return to the call site must also be trusted (or need not be logged). BLACKBOX instruments each call site with a push to a shadow stack, and links all returns to an in-cache assembly routine that pops from the shadow stack and verifies the destination.

Multiple Shadow Stacks

On the Windows platform, a system call may perform a callback to the program. Since these callbacks typically occur at the end of the system call implementation, it is common for the callback to occur on a chain of tail calls, such that the return from the callback goes directly to the original system call site. A naïve shadow stack will mistake this for an incorrect return, so BLACKBOX implements *nested shadow stacks* to account for the callback. Since BLACKBOX already intercepts all interaction between the monitored program and the operating system, there is no additional runtime cost to detect a callback and push a sentinel value that represents the opaque stack frames occurring within the system call implementation. Then, whenever the callback returns to the system call, BLACKBOX simply unwinds the shadow stack to the frame below the sentinel. This allows BLACKBOX to trust tail calls in callbacks without allowing the adversary to masquerade a return-oriented programming (ROP) attack by linking arbitrary return targets under the guise of chained callbacks.

Another popular programming idiom makes unconventional use of the `ret` instructions to make a context switch between virtual threads (known as a *soft context switch*). For example in Windows, the Fibers API stores several soft contexts in dynamically allocated memory and switches between them by copying the stored state into the physical CPU registers. Since each context has its own separately allocated stack, this switching process changes the stack pointer and corresponding return address. A naïve shadow stack will detect an incorrect return at every soft context switch. BLACKBOX avoids this by allocating a separate shadow stack for each soft context, and detecting a switch when stack motion exceeds the physical stack limit designated in the executable header. Shadow stacks are correlated to soft context stacks in a hashtable keyed by the stack base address.

3.3.3 Trusting Self-Instrumentation

In the Windows development ecosystem, it has become popular for programs to occasionally override functionality provided by major vendors such as Microsoft, perhaps because the centrality of these organizations precludes the development and/or distribution of alternative libraries. Since the functionality is provided in statically compiled binaries and linked by a name-based lookup at runtime, there is no systematic way to substitute these library functions. Instead, programs simply change the page permissions on the loaded image of the library and write a 5-byte `jmp` over the entry point of the unwanted function. This instrumentation, known as a *hook*, typically takes place during program startup, and often occurs after the original (un-hooked) version has already been executed from the BLACKBOX code cache. The Trusted Profile that worked so nicely in Paradise is confounded by this runtime transformation, and would cause an anomaly to be reported on every execution of the hook. To account for multiple versions of the same basic block, BLACKBOX adds a hash code of the basic block's instruction bytes to the Trusted Profile, along with a chronological version number.

3.3.4 Modular Trust

Today's software is typically composed of an executable and a collection of libraries that it loads at runtime. One of the key advantages to this approach is that an application can be compatible with several different versions of the same library. In Paradise, modules were always loaded at program startup, allowing for a single monolithic Trusted Profile that does not include an abstraction for modules. But since S_{cc} on x86 introduces dynamic module loading and unloading, it is necessary for the Trusted Profile to be segmented by module, and to link these Trusted Profile segments as the corresponding modules are loaded. BLACKBOX defines a *cross-module* edge type to represent function calls across modules (`jmp` edges across modules are never trusted).

One module may be compatible with several versions of a linked module, and it is even common for large Windows programs to load multiple versions of a module simultaneously. To allow sufficient flexibility in the Trusted Profile, the cross-module edge identifies its destination using the base name of the module and the export name of the callee, which effectively mirrors the Windows dynamic linking mechanism. If multiple versions of a module have been loaded simultaneously, this approach will alias the cross-module edges to the two versions. While this is slightly more flexible than necessary, it is unlikely that an adversary could use it for any significant advantage, given that a separate Trusted Profile guards each version of the module.

Callback Functions

Windows applications can use the callback programming pattern, which produces cross-module calls to private functions having no exported name. For these cross-module calls, the Trusted Profile uses the generic label `<foo!callback>` to represent any callback from a module `foo`. This effectively aliases all the targets of a callback site, adding even more superfluous flexibility to the Trusted Profile than the name-based cross-module edges. But since the aliasing is limited to callees that are trusted at the callback site, the opportunity for adversarial manipulation is extremely limited.

3.3.5 Detecting Evasion

One of the potential weaknesses for a virtualization-based security monitor like BLACKBOX is that a malicious process may be designed to detect the presence of security tools. A taxonomy of anti-virtualization and anti-debugging outlines the system artifacts that evasive malware commonly check when determining whether a machine is dedicated to security monitoring [26]. These factors cannot be determined from outside the machine without some cooperative agent running on the machine itself. Such an agent could take three forms:

A1 A program that is (a) designed to communicate information over an external access channel such as a network, and (b) has a bug allowing an attacker to obtain protected information from the machine.

A2 A standalone malicious executable.

A3 The payload of an intrusion that can take over a thread in any running program.

In all three cases, either A_{wpm} makes the attack vector detectable up to the limits of C_{prof} in the current BLACKBOX deployment, or the attack is outside the BLACKBOX threat model.

For case A1, if the program is not monitored by BLACKBOX then the attack is out of scope. Otherwise the adversary either leverages a control flow bug in the program to leak information, or abuses a faulty information flow channel. Information flow vulnerabilities are out of scope for BLACKBOX and better addressed by defenses focusing on that domain. Under G_{wx}, a control flow attack occurring in a program monitored by BLACKBOX should be detected and reported before information is leaked, even if the attack vector is unknown and unforeseen.

Case A2 is the responsibility of the system user. If processes on the system are able and willing to indiscriminately launch externally introduced executables, then the system has administrative vulnerabilities that are outside the scope of BLACKBOX. Similarly, if the system user, for example, receives an executable in an email attachment and runs it, that is a social engineering vulnerability that BLACKBOX cannot address.

Case A3 is similar to case A1: if BLACKBOX is monitoring the program, it should detect the intrusion under G_{wx}. Otherwise the attack is out of scope because BLACKBOX can only focus on one monitored process.

Unforeseen attack vectors may always exist, but within the corpus of known malware capabilities a successful detect-and-evade tactic must fall into one of these three cases. For example, advanced malware techniques for detecting honeypots involve scanning and even testing outgoing network connections, which requires a local agent [198]. Another example is evaluation of system age and usage patterns, which requires access to the filesystem, drivers, BIOS, installed programs, etc. [112]. Intuitively, for malware to investigate the environment from within a process monitored by BLACKBOX, then BLACKBOX should always see the malware before the malware can see BLACKBOX.

The attack vectors that are out of scope can be brought into the domain of BLACKBOX by simply running the Windows desktop in BLACKBOX. In this configuration, every launched process is automatically monitored under BLACKBOX before control enters program code.

3.4 Dynamically-Generated Code

The Trusted Profile and monitoring techniques for statically compiled code can be similarly applied to the monitoring of dynamically generated code, provided that the instruction content of the DGC is consistent between executions of the program. BLACKBOX distinguishes this kind of DGC by a property *cross-execution consistency*, which requires that for any two observationally equivalent executions of a program, the DGC component always has the same sequence of instructions. In the Windows development ecosystem, this property typically holds for simpler dynamic code generators, but to our knowledge never holds for the JIT compilers used by script interpreters. Random factors such as hot path tracing (which is affected by CPU thread scheduling) can cause the generated code to vary wildly, even between executions of a semantically trivial, single-threaded benchmark. The complications for BLACKBOX are compounded by S_{opq} vis-à-vis the interpreted script. Since it is not possible to create a reliable Trusted Profile for either the script or the JIT-generated code, the core IID monitoring technique cannot be used to determine whether the JIT is under the influence of the adversary. Even if there were another IID operating at the script level, the adversary could potentially leverage a bug in the application's interpreter to spray the JIT code with byte patterns that facilitate an attack within the JIT code. In this case, neither IID will be able to detect an attack that executes entirely within the JIT code region.

To avoid this vulnerability, BLACKBOX adds a set of edge types to the Trusted Profile that abstractly represent the behaviors of a code generator. When execution of a JIT application conforms to these Trusted Profile edges, BLACKBOX trusts that it is not under the influence of the adversary. Section 3.4.1 presents this *dynamic code permission model* in detail, and shows how BLACKBOX applies it both to code generators that have cross-execution consistency and those that do not. Section 3.4.2 follows with additional adjustments for dynamic code generators that do have cross-execution consistency. Although we were not able to find working versions of exploits against JIT engines, a case study in Section 3.4.2 shows how this permission model can detect one of the most challenging exploits developed in recent research.

3.4.1 Permission Model

Dynamic code generators in today's Windows programs have three common characteristics that form the basis of our permission model:

1. The memory in which the dynamic code resides is usually allocated by the module that generates the code.
2. Permissions for DGC memory pages are usually managed by a fixed set of call sites within the code generator.
3. There are typically fewer than 20 store instructions in the code generator that write to DGC memory allocations after they are granted `executable` permission.

While there may exist code generators that do not exhibit these characteristics, it is sufficient for the purposes of BLACKBOX that this set generally holds for the most popular JIT engines, including Mozilla Ion, Chrome V8, Internet Explorer's Chakra, and the Microsoft Managed Runtime.

The BLACKBOX dynamic code permission model adds three new edge labels to the CFG:

- **gencode write** between nodes X and Y indicates that an instruction in node X (or in a callee of call site X) wrote dynamic code Y, which was later executed.
- **gencode chmod** between nodes X and Y indicates that an instruction in node X (or in a callee of call site X) changed the `executable` permission on a memory region containing dynamic code Y, which was later executed.
- **gencode call** indicates an entry point into dynamic code from statically compiled code, or (similarly) an exit from dynamic code into statically compiled code.

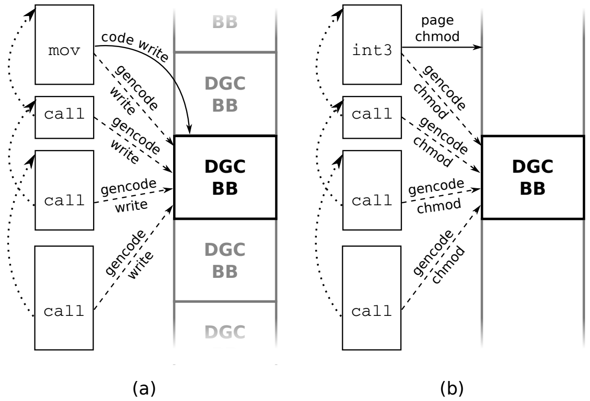


Figure 3.3: Construction of the (a) *gencode write* and (b) *gencode chmod* edges (dashed arrows) when a code write or page chmod occurs (solid arrows). The `call` sites represent the call chain of the code generator and its dependent libraries (upward dotted arrows).

Figure 3.3 illustrates the construction of the *gencode write* and *gencode chmod* edges. These edge types are complementary, such that for any DGC basic block, BlackBox observes at least one gencode write **or** gencode chmod, but not necessarily both. This guarantees visibility of DGC whether it is written directly to an `executable` region, or written to a buffer that is later made `executable`.

Since it is difficult for BlackBox to determine which call on the stack made the semantic decision to write DGC or `chmod` a DGC region, BlackBox simply creates an edge from every call site on the stack. There is one special case for a gencode chmod by a module that keeps its DGC private (i.e., no other module calls its DGC): BlackBox assumes the semantic decision came from that module, and only creates gencode chmod edges up to that module's call site.

BlackBox observes the three gencode actions during profiling and writes the edges to the Trusted Profile. This becomes the permission model for the program's dynamic code generators. As BlackBox monitors the program at the client site, gencode actions that conform to these permissions are considered safe, and logging is elided—but if the program takes any other gencode actions, those anomalies are logged to the IID Service. Section 3.4.2 gives a concrete example of both logging and blacklisting the pivotal attack vectors of a state-of-the-art JIT injection attack.

Observing Dynamic Code Writes

Observing the gencode write is challenging because the performance overhead of instrumenting all store instructions would be far too high. BLACKBOX takes an over-approximating approach by leveraging the operating system's memory permissions to observe all writes to memory pages that have ever been set `executable`. BLACKBOX maintains a shadow page table with a set of *potential code pages*, and adds any page to that set when it is first set `executable`. Whenever the monitored program sets a potential code page to `writable`, BLACKBOX artificially makes it `readonly`, such that any write to the page causes a fault that BLACKBOX intercepts and handles:

- Add a *potential code write* entry to the shadow page table for the specific range of bytes written.
- Change the memory permission to `writable` and allow the program to execute the write.
- Reset the memory permission to `readonly` so that future rewrites of the region will also be detected.

If the program's writes did contain code, BLACKBOX relies on the fact that it does not appear in the code cache yet: every time new code is cached from a dynamically allocated page, BLACKBOX consults the shadow page table, and if any *pending code write* entries are found associated with those fresh basic blocks, then BLACKBOX creates the corresponding gencode write edges—one from each call site that was on the stack at the time of the write. While it is possible for the program to write code and never execute it, BLACKBOX can elide the corresponding gencode write edges because code that is never executed can do no harm.

To mitigate the page fault overhead for store instructions that frequently write code (typical of JIT engines), BLACKBOX can instrument the instruction with a hook to create the gencode write edges. This approach greatly improves performance on aggressive JavaScript benchmarks [74].

Dynamic Singleton Node

Since BLACKBOX is often unable to correlate low-level control flow in dynamic code between executions of the program, each contiguous subgraph of dynamic code is represented as a dynamic singleton node. BLACKBOX establishes a *trusted vocabulary* for each dynamic singleton: for every *program action* taken within the dynamic code region, a self-edge of that type is added to the dynamic singleton. This allows BLACKBOX to take advantage of useful properties of popular JIT engines such as Microsoft Chakra and Chrome V8, which throughout our corpus of experiments never generate a gencode chmod self-edge. Anytime an executing thread takes a branch from one dynamic singleton to another, the two are merged by combining all their edges.

3.4.2 Standalone Dynamic Routines

BLACKBOX can be configured to log *standalone dynamic routines* in more detail than the coarse API logging of the dynamic singleton. We leverage two observations:

- For contiguous subgraphs of dynamic code having fewer than 500 basic blocks, the number of distinct permutations of these routines across program executions is relatively low, making the total size of all observed permutations small enough to fit in the Trusted Profile.
- Small contiguous subgraphs of dynamic code usually have an *owner*, which is a statically compiled module that takes exclusive responsibility for (a) writing its dynamic code, and (b) setting `executable` permission on the memory where its subgraph resides.

When standalone dynamic routine monitoring is enabled, BLACKBOX writes the CFG for every standalone to the Trusted Profile of its owning module. This approach adds a small overhead because BLACKBOX must check the Trusted Profile every time it copies new dynamic code into its code cache. But for all the programs we have observed, including frequent standalone generators like Microsoft Office, the program never modifies its standalones, making the overhead relatively insignificant.

Matching Standalone Dynamic Routines

Since dynamic code can be placed at any arbitrary memory location, and there is no module boundary to define a reliable relative address, BLACKBOX identifies each basic block in a standalone dynamic routine by (a) the hashcode of its instruction bytes and (b) its edge relationship with neighboring CFG nodes. When a new dynamic code entry point is observed, BLACKBOX creates a candidate list of standalones populated from the trusted profile. As new basic blocks are copied into the code cache, each candidate is checked for a corresponding node with the same hashcode and having the same edge relationship. Candidates having no match are removed from the list, and if the candidate list becomes empty, BLACKBOX marks the standalone as suspicious and logs its current (and any future) basic blocks to the IID Service.

If at any point the standalone dynamic routine takes an edge into existing dynamic code, BLACKBOX traverses the set of newly connected nodes until:

- The total size exceeds the (configurable) upper bound of 500 basic blocks; the two are combined into a single JIT region (creating a new one if necessary).
- All candidate routines are rejected; the new routines is logged to the IID Service as suspicious.
- The end of the connected region is reached; the new routine remains a potential match for its candidate routines.

This approach is advantageous for identifying code injection attacks, because the total size is often smaller than 500 basic blocks, such that the entire injection will be logged to the IID Service. Even if it exceeds this size, the standalone dynamic routine matching algorithm will progressively write its first 500 basic blocks to the IID Service as they are executed. The only ways for the adversary to hide such an injection are challenging in practice:

- Exactly match the instruction hash and edge relationships of an existing standalone dynamic routine (for every basic block), which exponentially reduces the attack options, or
- Enter the injection from existing JIT code, which effectively requires another code injection in the JIT region.

G_{rx} Case Study: Blocking Code Injections

Recent advances in browser security make it much more difficult for an attacker to gain control of compiled JavaScript. For example, the Chakra JIT engine in Internet Explorer places all JavaScript data in non-`executable` pages, and prevents the abuse of JavaScript constants by obfuscating them. While these techniques make it increasingly difficult to exploit the browser via JavaScript, [5] demonstrates a code injection that leverages ROP to compromise a recent version of Internet Explorer:

1. Coerce the victim's browser into loading crafted JavaScript.
2. Wait for the browser to compile the ROP payload.
3. Pivot the stack pointer via `xchg` to the phony ROP stack.
4. Execute the ROP chain, which invokes `VirtualProtect` on a page of memory containing injected shellcode.
5. Adjust the ROP chain to `ret` into the shellcode.

While we were not able to obtain a working instance of this exploit, it is still possible to show how BlackBox can detect and block it. To train the Trusted Profile of the Chakra JIT engine, the author of this paper used Microsoft Outlook for email during a 4-week period. The profile contains no edges from the Chakra dynamic singleton to system calls, and no self-edges of type incorrect return or gencode chmod. Suppose he now receives an email containing the crafted JavaScript. BLACKBOX will log several untrusted program actions:

- Steps 1 and 2 are transparent to BLACKBOX because they constitute normal execution of the JIT.
- At steps 3 and 4, BLACKBOX will log an incorrect return for each link in the ROP chain, because the dynamic singleton has no self-edge of type incorrect return.
- At the end of step 4, BLACKBOX will additionally log the system call to `Virtual Protect` because the dynamic singleton has no edges to any system calls.
- In step 5, a branch is taken into a new dynamic code region, causing it to be incorporated into the dynamic singleton. Since that new region was set `executable` by the dynamic singleton itself, BLACKBOX will log a gencode chmod self-edge to the IID Service.

The authors of this exploit claim that no existing security technique is able to detect it, much less stop it from taking full control of the browser. But the exploit can easily be blocked by BLACKBOX. Each of the program actions that are logged during this exploit are unique to malicious behavior—the Trusted Profile decisively indicates that Outlook would never take these actions outside the influence of crafted input—so blacklisting these actions will not cause any interruption in normal usage of Outlook. While BLACKBOX does require an expert to identify the pivotal attack actions, it is a relatively simple analysis.

3.5 Watchdog Mode

Since BLACKBOX monitors a program at a much higher granularity than Paradise, it requires more complete profiling coverage or will be susceptible to a high rate of spurious anomalies. This is not so much of a problem for G_{dbg}, where the goal is to find information associated with a specific incident, but greatly reduces the effectiveness of preventative monitoring. To improve detection of preliminary symptoms—such as a malware author's pre-release testing or even the voluntary use of a new plugin that has security vulnerabilities—BLACKBOX provides two techniques that highlight suspicious behavior in the presence of benign anomalies:

- *Stack spy* separately logs any system call that occurs in a suspicious stack context.
- *Sniffer dog* uses PowerLaw modeling to identify log entries having a suspicious smell, which can be defined as an outlier along the spectrum of any security sensitive control flow property.

These techniques can significantly narrow the focus of routine security audits, as demonstrated by the experiments reported in this section.

3.5.1 Stack Spy

The BLACKBOX stack spy leverages the insight that the greatest risk to the security of a monitored program occurs along the control flow paths to system calls. A typical ROP attack takes control of the instruction pointer and drives execution through shellcode that is completely foreign to the victim program, with a goal of executing system calls to access the file system and/or network. More sophisticated attacks employ crafted input to cause a slight detour along the program's normal route to a system call, enabling the attacker to modify the effect of that system call for malicious advantage. In most such cases, BLACKBOX will observe at least one untrusted branch along the control flow path to the system call. To isolate this scenario, the stack spy separately logs suspicious system calls that occur while any frame on the call stack has been influenced by an untrusted branch.

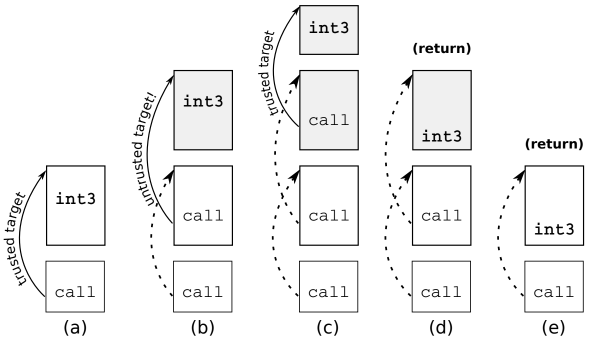


Figure 3.4: System calls occurring under stack suspicion (gray) are logged even if the syscall site is trusted. Stack spy raises suspicion in the stack frame where an untrusted program action first occurs (b), and clears suspicion when that stack frame returns (e). The syscalls in (c) and (d) cannot be elided because stack suspicion is inherited by callees, but the syscalls in (a) and (e) may be elided because suspicion has not yet been raised (a), or has been cleared (e).

Stack spy implements this feature using a simple boolean flag for each program thread, as illustrated by the function boxes in Figure 3.4. The flag is initially false (white), and when an untrusted branch occurs (step b), *stack suspicion* is raised at the current stack level (i.e., `esp` in x86 platforms). Any system call made under stack suspicion (gray) is logged to the IID Service along with the untrusted branch—even if the system call itself is in the Trusted Profile (b, c and d). When the thread eventually returns from the stack frame in which the untrusted branch occurred (e), stack suspicion is cleared, and future system calls on that thread can again be trusted.

G_{dnd} Case Study: Authoring Tools

The author of this paper used the SciTE text editor and MikTek pdflatex under watchdog mode while writing this research paper and developing the gencode write and gencode chmod features of BLACKBOX. The Trusted Profile for each program was trained during the first half of the experiment, and remote logs were accumulated during the second half, as shown in Table 3.2. More than 100 untrusted indirect branches are logged per day, which represents a significant workload

Logged Program Action	SciTE	pdflatex
Indirect Branch	132	9
Suspicious Syscall 0x25	1	0
Suspicious Syscall 0x47	1	0
Suspicious Syscall 0x52	3	0

Table 3.2: Log entries per day while writing the BLACKBOX publication under BLACKBOX. Stack Spy highlights any questionable filesystem syscalls (**`0x25`** `NtMapViewOfSection`, **`0x47`** `NtCreateSection`, **`0x52`** `ZwCreateFile`).

for preventative analysis. But the rate of suspicious system calls is fewer than 5 per day, making it possible to efficiently verify that the file system activity from these two programs is not under the influence of malware.

$G_{\mathbf{rx}}$ Case Study: Detecting COOP Attacks

Counterfeit Object-Oriented Programming (COOP) [149, 34] is designed to thwart control-flow integrity (CFI) schemes: by injecting the target program with bogus objects having crafted virtual dispatch tables, this exploit deviously conforms to category-based CFI policies that only constrain the protected program to make method calls at method call sites. The BLACKBOX Trusted Profile, however, does not contain phony branch targets, so the hijacked calls in a COOP attack will be logged to the IID Service as they occur.

While it would be ideal to blacklist all indirect branch targets not appearing in the Trusted Profile, this is not generally possible—our experiments show that in large Windows programs, normal new branch targets do occasionally occur for known indirect branches. But for a given COOP attack, the BLACKBOX stack spy can isolate branches leading to the system calls that comprise the payload, making it relatively easy for the log analyst to blacklist the pivotal attack points.

3.5.2 Sniffer Dog

In addition to spying out the most suspicious system calls, watchdog mode provides an offline sniffer dog that sorts the most suspicious smelling program actions to the top of the log. Sniffer dog employs a principle of "typical irregularities" to estimate the probability that an untrusted program action is a safe variant of trusted behavior. Examples of especially suspicious smelling behavior are (a) the addition of a second target to an indirect branch site having only one trusted target, and (b) the addition of a cross-module branch between two modules having no trusted edges between them. The iterative process of Trusted Profile training reveals how frequently new edges are normally discovered in each region of the CFG. For example, during profiling of Google Chrome, new indirect branches (and branch targets) are routinely discovered within `chrome_child.dll`—even during the final iterations—since it is a very large module providing a diverse set of features. In contrast, profiling of IISExpress on both static HTML and WordPress (PHP) rarely encounters new edges in the main module `iisexpress.exe`, since its role is limited to server startup and simple routing of requests.

To concisely capture these observations, BLACKBOX records a history of new edge discovery between each possible pairing of modules (reflexive included), and summarizes each with a PowerLaw model [3]. Sniffer dog consults these models while sorting the log to determine which entries most contradict the typical behavior of the program; log entries conforming to the PowerLaw models are given lower priority, while those exceeding the model's prediction for new events are given higher priority. Even if the adversary is aware of this modeling approach, it still creates significant limitations, mainly because the module connectivity of the large Windows programs in our experiments is extremely sparse—most pairs of modules have no edges between them. This appears to hold true for many large desktop programs, which are constructed in component hierarchies where modules at each level interact only with modules at the same and adjacent levels. For any arbitrarily selected gadget that may be useful to the adversary, there is a high probability that it will create an edge that doesn't fit the PowerLaw model, such that Sniffer Dog will assign it high suspicion.

$G_{\textbf{mod}}$ $G_{\textbf{dbg}}$ Case Study: ROP exploit in Adrenalin

Listing 3.1 shows a sorted log for the Adrenalin Media Player in typical usage. The five suspicious edges are all typical of Adrenalin, so Sniffer Dog assigns them a low rank of 396. BlackBox does not attempt to conclude whether malicious influence occurred in an execution, but even if an exploit did occur on the day this log was generated, the low suspicion rankings along with the complete absence of suspicious system calls would strongly suggest this execution was not involved.

Listing 3.1: BlackBox log for typical usage of Adrenalin. A few suspicious anomalies occur, but at a very low suspicion ranking (396).

```
396 Suspicious indirect play.exe(0xca1f0 → 0x22f90)
396 Suspicious indirect play.exe(0xca1f0 → 0x22f30)
396 Suspicious indirect play.exe(0xc2358 → 0x20870)
396 Suspicious indirect play.exe(0xc2358 → 0x218c0)
396 Suspicious indirect play.exe(0xc2307 → 0xa1b50)
127 Structural indirect adrenalinx.dll(0x96400 → 0xe6770)
069 Structural indirect play.exe(0x22fa5) → Lib(0x4ad91c)
069 Structural indirect play.exe(0x22f45) → Lib(0x4ad91c)
```

Listing 3.2 shows the anomalies that occur while the Adrenalin Player processes a video in a format that was not encountered during profiling. Two of these anomalies are raised by Stack Spy to the high rank of 900 because the filesystem was accessed on a call stack having a suspicious branch. The system call `NtSetInformationFile` is capable of many potentially dangerous operations such as truncating files, renaming files, creating hard links in the filesystem, and a variety of other potentially dangerous operations. Although there were no anomalies related to creating file handles during this execution, it would still be possible for an adversary to damage the system by manipulating the arguments to just one of these system calls. Fortunately, BlackBox can log all arguments to any suspicious system call, which in this case would resolve any question about malicious influence. For example, if the calls simply modify the `FilePositionInformation`, the user can be sure that no harm was done. However, if the suspicious calls were renaming files, then it would be essential to know the source and destination paths, which are provided by BlackBox when deep argument logging is enabled.

Even if there is no malicious influence in this execution, the user still may find it valuable to know that these anomalies occurred. For example, a quick web search might reveal that Adrenalin does not handle this video format correctly. In this case the user could simply avoid those files in Adrenalin, or go so far as to blacklist the incorrect control flow. It is not quite as simple as blacklisting the edge to the system call, since that edge is used on many other control flow paths and is essential for accessing the file system or the network. But the branch anomalies reported in the suspicious system call are likely to be unique to the suspicious behavior, so the user can simply copy those log entries into the blacklist (with the help of a BlackBox utility).

Listing 3.2: BlackBox log of Adrenalin handling a format variation not encountered during Trusted Profile training. Sniffer Dog assigns a high degree of suspicion (900) to the untrusted indirect branches associated with this untrusted file format.

```
900 Suspicious syscall #36 NtSetInformationFile
    adrenalinx.dll(0xf160a → 0x97ac0) raised suspicion
900 Suspicious syscall #36 NtSetInformationFile
    adrenalinx.dll(0xf160a → 0x97ac0) raised suspicion
300 Structural indirect mp3dmod.dll(0xe342) → Lib(0x4be75b)
300 Structural indirect Lib(0x2f3994) → addicted.ax(0x39280)
300 Structural indirect mp3dmod.dll(0x5800 → 0x59ed)
300 Structural indirect mp3dmod.dll(0xe37a) → Lib(0x4be75b)
295 Structural indirect Lib(0x9a7beb) → qasf.dll(0x2a7e9)
295 Structural indirect qasf.dll(0x2c86e) → Lib(0x13f853)
295 Structural indirect qasf.dll(0x28a2f) → Lib(0x13f853)
... (many similar entries)
```

Listing 3.3 shows the anomalies that occur in Adrenalin during an ROP exploit, which launches `calc.exe`. All of the highest-ranking anomalies are specifically highlighted by the Watchdog Mode features. Sniffer Dog identified the top 8 as extremely improbable for Adrenalin, and Stack

Spy identified the following suspicious system calls. References to "DGC" indicate the payload, which is deployed by the exploit as a code injection via buffer overflow. When BLACKBOX is configured to generate a complete trace after a high-suspicion anomaly, the entire sequence of control flow edges is logged in chronological order, starting with the incorrect return from `adrenalinx.dll`. Combined with a trace of system call arguments, the full BLACKBOX trace would clearly show how the exploit took control of the program, and what malicious actions were taken in the payload.

Listing 3.3: BLACKBOX log of Adrenalin during an exploit. Sniffer Dog assigns the highest suspicion level (999) to the pivotal control flow edges of the exploit.

```
999 Suspicious entry into DGC
    adrenalinx.dll(0x16f313) → Lib(0x3bfffb) raised suspicion
999 Incorrect return adrenalinx.dll(0x16f313) → Lib(0x3bfffb)
999 Untrusted module calc.exe-1db1446a00060001
999 Suspicious indirect shlwapi.dll(0x1c508) → Lib(0x192aa7)
999 Suspicious indirect ntdll.dll(0x3c04d) → Lib(0xe2e831)
999 Untrusted module gdiplus.dll-1db146c800060001
999 Suspicious indirect kernel32.dll(0x13365) → Lib(0xdb3fc3)
998 DGC standalone owned by adrenalinx.dll-300010001 (4 nodes)
900 Suspicious syscall #25 ZwSetInformationProcess
    ntdll.dll(0x22373 → 0x224b0) raised suspicion
900 Suspicious syscall #82 ZwCreateFile
    ntdll.dll(0x2239c → 0x22468) raised suspicion
900 Suspicious syscall #79 ZwResumeThread
    kernelbase.dll(0x14148) → Lib(0x6ee3a) raised suspicion
900 Suspicious syscall #26 ZwCreateKey
    user32.dll(0x16d88) → Lib(0xd5f105) raised suspicion
900 Suspicious syscall #77 ZwProtectVirtualMemory
    apphelp.dll(0x13066) → Lib(0xd5f105) raised suspicion
900 Suspicious syscall #165 ZwCreateThreadEx
    kernelbase.dll(0x13f6d) → Lib(0xa57647) raised suspicion
... (flood of similar entries)
```

The chronological trace also makes it relatively easy to blacklist the exploit to prevent recurrence—it is simply a matter of copying the most suspicious edges into the blacklist. This corresponds to prohibiting the following program behaviors:

- Executing a `ret` from `adrenalinx.dll(0x16f313)` in the vulnerable function to `Lib(0x3bfffb)` (whether by overwriting the return address or any other means).
- Executing any of the edges listed as `Suspicious indirect` or `Suspicious entry into DGC` (whether invoked by JOP gadgets or any other means).
- Loading the module `calc.exe`.
- Any DGC owned by module `adrenalinx.dll-300010001`.

It is certainly possible for an application to use these exact behaviors in a normal way, and in that case the blacklist entries would interfere with normal program behavior. But the fact that BLACKBOX assigned the highest suspicion indicates that (a) the program was not known to use these behaviors normally, and that (b) the probability is extremely low that it would ever do so. Manual analysis confirms that Adrenalin will never take these actions normally.

A security expert could improve protection against this exploit by generalizing the first three blacklist entries to prevent similar exploits as well:

1. Making any incorrect return from `adrenalinx.dll(0x16f313).`
2. Executing an untrusted edge from any source node of the `Suspicious indirect` entries.
3. Loading a main executable image as a module.

The first generalization would effectively protect the buffer overflow vulnerability itself, such that no attack against that particular buffer would be able to initiate an ROP chain. Source nodes referred to by the second generalization are jump-oriented programming (JOP) gadgets, so the corresponding blacklist entries would take those gadgets away from the adversary. The third generalization is nearly universal—a user might be tempted to apply it to all programs, not realizing that there are a few cases where a program normally does this. For example, consider a utility program that can also be used as a plugin for some larger application. A universal policy would block that plugin from being loaded, but the context-specific policy (shown here) will not, so that is probably the better choice.

3.6 Evaluation

This section begins by presenting quantitative experiments with BLACKBOX to evaluate its effectiveness towards four of the IID goals. Section 3.6.1 pursues G_{dnd} using a variety of controlled and real-world experiments focusing on typical Windows applications along with the IIS webserver and PHP. In Section 3.6.2, BLACKBOX identifies and then blacklists three known exploits against small Windows applications to fulfill G_{kx} and G_{blk} while confirming G_{log}. Section 3.6.3 pursues G_{perf} with an IIS workload along with the standard SPEC CPU 2006 benchmarks, and concludes with qualitative discussions of the development cost and deployment effort of BLACKBOX. Section 3.6.4 concludes the evaluation with a case study in support of G_{dbg}, showing how BlackBox's visibility into cross-module branches can be an essential complement to many important defense techniques.

Case studies were presented Sections 3.4 and 3.5 to demonstrate the effectiveness of BLACKBOX towards G_{rx} because it was not possible to obtain a working version of the necessary exploits, and G_{mod} and G_{dbg} can be evaluated on the basis of the Adrenalin logs and blacklist in Section 3.5. A statistical analysis of BLACKBOX security in Section 3.6.4 focuses on its utility for G_{dbg}. We did not attempt to evaluate BLACKBOX on G_{wx} because, while it is feasible for an automated script such as a browser robot to encounter a browser exploit in the wild, the probability is too low for the practical limitations of research experiments.

We conduct the quantitative experiments in Windows 7 SP 1 running in VirtualBox 4.2.10 on an Ubuntu 13.04 host using an Intel Xeon E3-1245 v3 CPU. The Windows Update service and application updates are disabled to maintain consistency throughout the experiments.

3.6.1 Filtering Log Noise

Given the perfectly complete Trusted Profile that we had in Paradise, BLACKBOX would generate no spurious anomalies because all the observed control flow events would be trusted. Our experiments show that, in the presence of S_{cf} and S_{sem}, it is not easy for BLACKBOX to obtain such complete coverage when profiling large applications like Microsoft Office. But for reasonable training periods with just one user, coverage can be sufficient to achieve a low rate of spurious anomalies that are rarely assigned high suspicion.

Configuration for Interactive Applications

In the typical usage scenario for IID, the vendor of an application will generate the Trusted Profile using internal facilities associated with development of the product. BLACKBOX can also be to monitor legacy applications where the development infrastructure is not available, but this requires establishing a profiling procedure specific to the application. Since it was not within the scope of this research project to evaluate the vendor profiling scenario, these experiments focus on both profiling and monitoring.

For legacy applications having significant interactive components, the ideal evaluation of G_{dnd} would be a large user study involving people from various backgrounds whose daily work relies on a popular and complex Windows application. After recording execution traces over a long period of time, an offline analysis would determine the cost/benefit of various profiling durations. This would also facilitate evaluation of collaborative profiling strategies in which Trusted Profile content is shared among groups of similar users having a substantial basis for mutual trust (for example, employees of the same company).

To simulate such a study, we divide the goals into two separate research questions that can both be evaluated by profiling just one user at a time. Taken together, the results of these sub-experiments indicate the potential for BLACKBOX to obtain sufficient profiling coverage in real-world usage of these applications.

- **Controlled Usage**:
  - *Procedure:* One user is given a task such as replicating the first page of a conference research paper in Microsoft Word. Each session includes the creation of one document, revising it to correct errors, saving it to disk, and exiting the program. The user is instructed to explore all the input widgets and output channels available in Word—such as the ribbon menu, context menu and shortcut keys, along with local files, cloud storage accounts and printers—but to avoid any program actions that are not directly related to creating and saving the document.
  - *Research Question:* Can sufficient coverage can be obtained for a limited set of features by profiling just one user who exercises only those features for a short period of time?
- **Real-World Usage**:
  - *Procedure:* One user continuously profiles an application under normal daily usage for several weeks. A representative subset of application content is reserved for a designated test period which occurs at the end of the experiment. Normal usage of the application continues through the test period, and the user additionally accesses the reserved content.
  - *Research Question:* Can sufficient profiling be obtained with just one user?

To establish a substantially complete profile of basic application functionality, each experiment started by profiling the application with a screen robot. For example, thousands of files were randomly downloaded and viewed in Microsoft Word and PowerPoint, and Adobe PDF Reader (it is possible for these files to be malicious, we assume a similar set of known-safe files could be obtained, for example from vendor test suites). In the controlled usage experiments, the profiling period included manual viewing or creation of a set of N documents, followed by a test period of another N documents having different content:

- Replicate the first page of 60 Usenix Security papers in Word (30 profiling, 30 test).
- Replicate a table of crime statistics for 30 of the United States published by the FBI, entering formulas for all cells that could be locally computed (15 profiling, 15 test).
- Create 50 short PowerPoint presentations from blog posts on software security, each including several formatted bullet lists and an image downloaded through the built-in Bing image search component. The theme for each presentation was selected through the built-in online theme browser (25 profiling, 25 test).
- View 200 Word documents downloaded from the U.S. Center for Disease Control, scrolling through each document manually (100 profiling, 100 test).
- View 200 nutrition and health lectures in PowerPoint downloaded from the U.S. Food and Drug Administration (100 profiling, 100 test).
- View 200 randomly downloaded manuals for security products in Adobe PDF Reader, scrolling through each document manually and then uploading it through the built-in cloud account browser (100 profiling, 100 test).

The real-world usage experiments included the following applications and input content:

- **Google Chrome**:
  - *Profiling:* Use the browser for ordinary research activities such as (a) finding and viewing research papers and technical articles, (b) personal activities such as Facebook, online shopping, and viewing YouTube videos and animated graphic art (including ebizmba.com, threejs.org, backbone.js).
  - *Test:* vimeo.com, a videosphere [65], a dynamically textured video [9], newegg.com and sections of the graphic arts sites that had been reserved for the test.
- **Microsoft Outlook**:
  - *Profiling*: Ordinary email usage in HTML format, with all images and scripts enabled, and over 100 filters configured to automatically sort incoming messages into folders. The LinkedIn and Facebook plugins were installed to display related content for each message, and the user subscribed to email lists featuring graphics-intensive advertising from major retailers.
  - *Test:* Continue similar usage, subscribing to additional graphics-intensive promotional lists.
- **SciTE and pdflatex**:
  - *Profiling:* Writing sections 1-5 of the first submission of the BLACKBOX publication and implementing the gencode write and gencode chmod edges for BLACKBOX.
  - *Test:* Completing the BLACKBOX submission, including implementation of visualization components for the IID Service that were used to generate figures, and implementing the gencode call edge for BLACKBOX.

$G_{\textbf{dnd}}$ Result for Interactive Applications

The final column of Table 3.3 presents the number of anomalies reported during the test period of each application, normalized to an hour of application usage by one person. None of these reports is ranked at high suspicion, indicating that although Trusted Profile coverage is not perfect, the application behaviors under test are effectively trusted. The first column of this table shows the vast number of anomalies that would be reported by a direct port of Paradise to the x86 platform. Successive columns show the reduction in log noise provided by each component of BLACKBOX that was not necessary in Paradise:

Unique Branches: binary translation filters recurring edges.

Unique Indirects: offline analysis filters direct branches.

Forward Indirects: the shadow stack filters normal (correct) returns.

Untrusted Indirects: the Trusted Profile includes intra-procedural forward indirect branches.

A small fraction of the reported noise would also be filtered by a direct port of Paradise, but the majority is intra-procedural control flow that the Paradise components did not need to account for. Although it may seem that dozens of anomalies per hour could make the log unusable, the vast majority of these anomalies are labeled by Sniffer Dog as *structural indirects*, indicating that the target is selected from a table held in read-only memory (and that the application has never changed the page permissions). A common example is a compiled `switch` statement. Given the goal of IID to detect any untrusted behavior, these are important anomalies to report—but since a limited set of statically valid targets is built into the branch structure, any context-insensitive CFI system would consider these edges to be completely safe.

Table 3.4 presents the per-user-hour anomalies for the incorrect return and gencode edges during the same application test periods, both with the Trusted Profile (“Before”) and without it (“After”).

The high number of total edge occurrences ("Before") indicates that these behaviors are common for many of these applications, but that the usage is consistent enough for BLACKBOX to learn the vast majority ("After"), even during these limited experiments.

Some important observations can be drawn from the results of this experiment for each of the proposed BLACKBOX usage scenarios:

- **Security debugging** can most immediately benefit from an instance of BLACKBOX at its current level of engineering because (a) resources will be available to largely automate the profiling process, and (b) the low rate of spurious anomalies poses little distraction for highly skilled and informed developers who may in many cases find these special cases in execution to be interesting for other software engineering purposes.
- **Forensic analysis** can also make practical use of this BLACKBOX implementation, though spurious anomalies may potentially increase the workload of routine audits.
  - **Vendor profiling** will inevitably improve the efficiency of this use case.
  - **Legacy profiling** may be a complicated and time-consuming task, although the legacy scenario has the advantage that the Trusted Profile remains relatively constant over time, since the core application binaries are fixed.
- **End-user deployment** will likely require significant support infrastructure, both for obtaining a usable Trusted Profile and for monitoring the BLACKBOX log. However, given adequate resources, the blacklist may prove effective for mitigating a known exploit or other security hazard until a patch can be obtained and installed.

Configuration for Server Applications

Although vulnerabilities can arise from S_{opq} at the interface between layers of abstraction, the same phenomenon can also improve the resilience of the Trusted Profile to changes in usage patterns,

Program	All Branches	Unique Branches (+ Cache Branches)	Unique Indirects (+ Analyze Directs)	Forward Indirects (+ Shadow Stack)	Untrusted Indirects (+ Learn Indirects)
Chrome	485,251,278,660	42,957,575	16,537,926	6,137,106	7
Adobe PDF	34,075,711,128	15,579,901	6,325,821	2,292,342	4
Word	603,491,452,236	14,589,337	2,590,444	580,655	24
PowerPoint	251,845,377,624	27,839,593	1,848,681	1,335,817	50
Excel	198,427,776,372	14,810,205	2,389,208	561,401	28
Outlook	547,678,615,056	24,121,810	2,375,352	615,708	4
SciTE	61,325,719,872	2,463,871	372,445	124,013	33
pdflatex	23,504,352,560	1,790,288	278,726	64,290	43
Notepad++	129,695,545,404	8,400,249	1,732,147	589,155	24
Adrenalin	48,881,533,212	3,024,797	1,159,407	791,847	603
mp3info	2,080,031,200	94,804,000	18,713,600	4,339,200	3

Table 3.3: Average number of log entries during an hour of normal program activity for progressive implementations of BlackBox. The first column represents a direct port of Paradise to the x86 platform, and the last column includes all components of BlackBox that were not necessary in Paradise (lower is better).

	incorrect return		*gencode chmod*		*gencode write*	
Program	Before	After	Before	After	Before	After
Chrome	3,957	1	3,473	1	6,532	1
Adobe PDF	1408	0	6,119	0	437	0
Word	671	0	2,767	3	24	0
PowerPoint	767	2	6,718	5	46	0
Excel	782	0	1,806	1	23	0
Outlook	2,304	1	1,149	1	48	1
SciTE	2	2	6	0	2	0
pdflatex	0	0	0	0	0	0
Notepad++	24	2	69	0	23	0
Adrenalin	4	1	378	1	21	1
mp3info	0	0	0	0	0	0

Table 3.4: Average number of log entries with and without a Trusted Profile for an hour of normal program activity (lower is better).

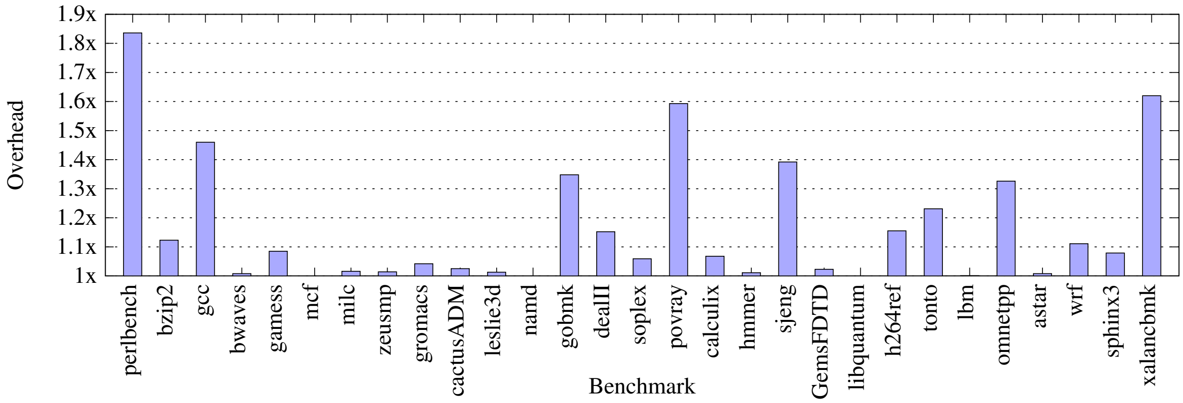


Figure 3.5: Normalized BLACKBOX execution times for Spec CPU 2006, taking the geometric mean of 3 runs (lower is better).

for example when monitoring a lower-level runtime such as a script interpreter. This experiment began by profiling IIS Express 7.0 with PHP 5.4 while serving typical web content: a 2GB snapshot of cbssports.com, the PHP unit test suite, and a default install of WordPress. To create changes in the usage patterns of the PHP interpreter, we performed several WordPress upgrades: installing the popular e-commerce plugin WooCommerce, activating the anti-spam plugin Akismet, changing the theme, upgrading the WordPress core, and importing posts into the blog and products into the store. Then we enabled the BLACKBOX monitor and ran a form-enabled crawler on the upgraded WordPress site for 6 hours, sending a minimum of 20,000 unique requests to each major area of the site: (1) store configuration and product editing, (2) blog administration and post editing, (3) theme customization, (4) public pages, and (5) upload forms.

$G_{\mathbf{dnd}}$ Results for Server Applications

Despite having added major functionality outside the scope of the Trusted Profile, BLACKBOX only logs 39 indirect branch anomalies (Table 3.5).

Program Action	PHP	IIS
Indirect Branch	33	6
incorrect return	0	0
gencode chmod	0	0
gencode write	0	0

Table 3.5: Total log entries during 6 hours of fuzz testing WordPress on IIS (lower is better). The Trusted Profile was trained on a default WordPress installation, but fuzz testing was executed after two plugins were installed, the theme was changed, and the WordPress core was updated.

3.6.2 Logging and Blacklisting Exploits

Three published exploits were executed against programs monitored by BLACKBOX to verify that (a) each program action induced by the exploit is logged to the IID Service, and (b) control flow stops at blacklisted program actions. It was not possible to obtain working exploits for the more popular Windows programs, due to bounties paid by vendors to prevent distribution or replication of those attacks, so these experiments focus on simpler programs.

Configuration for Known Exploits

This experiment used the same profiling procedure as the previous one for noise reduction. More specifically:

- **OSVDB-ID 104062 · Notepad++** We trained BLACKBOX to recognize Notepad++ and the vulnerable CCompletion plugin during a one-day development project to build a 500-line graphical chess game. After deploying the Trusted Profile, we continued development for two hours.
- **OSVDB-ID 93465 · Adrenalin Player** We trained BLACKBOX to recognize the Adrenalin multimedia player by opening and modifying dozens of playlists, and playing 100 mp3 files. After deploying the Trusted Profile, we continued similar usage of the player for two hours.
- **CVE-2006-2465 · mp3info** A script trained BLACKBOX to recognize the mp3info utility by executing 7,000 random commands on 300 mp3 files. After deploying the Trusted Profile and adding 50 more mp3 files to the experiment's corpus, the script executed 500 similar commands.

Starting with this configuration, the following goals were evaluated.

$G_{\textbf{kx}}$ $G_{\textbf{blk}}$ Detecting and Blocking Known Exploits

Upon deploying the Trusted Profile in each experiment, we configured the blacklist to pause with a dismissable warning, then attempted the exploit 3 times—each time with a different blacklist entry: the incorrect return where the exploit initiates the takeover, an untrusted indirect branch during the takeover sequence, and a suspicious system call in the exploit payload. No spurious anomalies occurred during normal usage of the programs, but the warning promptly appeared each time we executed the exploit. The exploit was not able to execute any system calls before the warning appeared, even for a blacklist entry on a suspicious system call (it appears just before the call). This indicates a termination action would prevent these exploits from doing any harm to the host machine.

$G_{\textbf{kx}}$ $G_{\textbf{log}}$ Detecting and Logging Known Exploits

To evaluate BLACKBOX logging, we disabled the blacklist and invoked the exploit again. As expected, BLACKBOX logged the incorrect return followed by a sequence of untrusted indirect branches and suspicious system calls that forked the payload (see Listing 3.3 in Section 3.5.2 for the Adrenalin log).

3.6.3 Resource Efficiency

For hardware-based execution environments, the inconvenience cost of the IID runtime is likely to always be a significant concern. This section presents the overhead of the BLACKBOX runtime, including a subjective evaluation of user experience, followed by a discussion of deployment and development costs.

$G_{\textbf{lite}}$ Runtime Performance

We measured the overhead of BLACKBOX on IIS with static content and obtained normalized execution times of .98 relative to native performance. The speedup is due to profile-guided trace optimizations in BLACKBOX. It is non-trivial to measure the overhead of BLACKBOX on the full IIS/PHP stack because execution-time overhead can be overshadowed by time spent waiting and by unimpacted calls like OS and I/O.

To more accurately understand the execution overhead, we employ industry standard benchmarks that focus on compute performance. We evaluated the performance of BLACKBOX relative to native execution on the SPEC CPU 2006 benchmark suite [75], which consists of a diverse set of CPU bound applications across several application domains and languages: 7 C++ programs, 12 C programs, 4 C/Fortran programs, and 6 Fortran programs. Across the suite of benchmarks we measured a geometric mean of 14.7% slowdown; Figure 3.5 presents the individual overheads. Benchmarks having mostly direct branches incur minimal (or zero) overhead in BLACKBOX, while programs having a greater proportion of indirect branches with multiple targets (typically C++ programs and script interpreters) incur the higher overheads. Profile-guided optimizations for high-degree indirect branches show promising results, but we reserve the formal evaluation for future work. While some of the larger Windows programs in our experiments do have a high rate of indirect branches, the majority are cross-module function calls through the IAT that by construction have only one target, making it feasible for BLACKBOX to optimize them as direct branches.

The BLACKBOX runtime has significantly higher overhead for programs such as web browsers that continuously generate large amounts of code. This is due to limitations in binary translation that affect the underlying DynamoRIO framework. Chapter 4 presents an optimization that improves performance enough to maintain a reasonable user experience for these applications. A limited version of this optimization was implemented to improve usability of DGC-intensive programs like Chrome and some components of Microsoft Office. While no formal benchmarks were attempted, the score for the Octane JavaScript benchmark suite [23] in BLACKBOX was between 4× and 5× the native execution score, which is roughly a 3× speedup over default DynamoRIO.

While it is difficult to quantitatively evaluate the effect of runtime overhead on user experience, the real-world experiments focusing on noise reduction (Section 3.6.1) provide a substantial basis for a qualitative evaluation. In each experiment, the user noticed significantly longer application startup times compared to native execution of the same application. There were also noticeable delays when loading JIT-intensive components in Microsoft Office and Adobe PDF Reader, such as the Bing image search panel and the Adobe cloud storage browser. But once these applications and components reached steady state, users reported no difference in the interaction response time for the majority of ordinary operations—including substantially graphical editing such as resizing and rotating groups of images, vector graphics and text in the Office "Smart Art" feature.

Sampling One approach to minimizing the overhead of BLACKBOX is to activate it intermittently, perhaps rotating across a group of individual users, such that no user is constantly experiencing its effects. This approach has been effective in similar security tools that focus on dataflow [72, 71, 92]. For long-running applications, it is possible to implement an attach/detach mechanism that would support sampling within a single program execution. If sufficient coverage can be obtained with a user-friendly sampling rate, the chances are still high that in the case of a widespread attack, some active BLACKBOX instance will record an attempt. The most significant problem with this approach, however, is that it gives the adversary a much greater opportunity to evade or even disable BLACKBOX without being detected. One of the fundamental concepts of IID is that the majority of adversarial tactics require some preparation, such as taking control of a thread to establish a side channel—if the IID is always present, the side channel cannot be used to evade the IID, because the IID is watching while the exploit tries to establish the side channel. But a sampling approach means turning a blind eye for a long period of time on any given machine, which may allow a very careful adversary to detect the presence of an active BLACKBOX and "play innocent" until the coast is clear. The feasibility of such an attack in practice remains to be seen, but there can be no question that it is theoretically possible.

G_{dep} Deployment

BLACKBOX can be installed on a Windows desktop machine with a conventional application installer, although we did not include this in our prototype. An administrative tool can be included in the installation to assist the user in creating application launchers in the Start Menu or on the Desktop. In our experiments, we launched applications in BLACKBOX using a simple shell command:

```
bb /app/Google/Chrome/Application/chrome.exe
```

Independent third-party reviewers were able to launch applications in BLACKBOX this way without our assistance.

End User Profiling For scenarios in which the profile will be generated by end users, it is recommended to use the IID Service, which can securely upload traces merge them into the Trusted Profile. The service can continuously estimate the coverage ratio as profiling progresses, on the basis of (a) the convergence rate of newly visited control flow edges, (b) the number of potentially unvisited paths, determined for example by symbolic execution (locally constrained to maintain accuracy and performance), and (c) the degree of coverage obtained by other IID instances for the same (or similar) application modules. The IID Service can potentially automate the entire process of profiling and logging, even under the complications created by the x86 platform.

$G_{\mathbf{dev}}$ Development

Another kind of overhead is the cost of developing and maintaining BLACKBOX, which includes a complex runtime along with several profiling and analysis tools. The BLACKBOX runtime itself is implemented in roughly 10,000 lines of code, and the supporting framework DynamoRIO is implemented in 250,000 lines of code, all in C89. Since DynamoRIO is a sponsored open source project, its support for the latest x86 ISA, as well as updates to Linux and Windows, is maintained by developers at Google. BLACKBOX integrates a security API into the DynamoRIO core, but this mostly affects code locations that are fundamental to the runtime and rarely change. The experiments reported here focus on some of the largest and most complex end-user applications for the Windows platform, suggesting that BLACKBOX is substantially complete, and development of an industrial version could likely proceed without major restructuring of the platform. The BLACKBOX profiling and analysis tools are implemented in under 20,000 lines of Java code and would only need to be updated in the case of architectural changes to BLACKBOX. Since these tools are shared among all users of the platform, and require no application-specific investment, we anticipate that these reasonable costs could be amortized by an increasing user base.

3.6.4 Security Analysis

A recent survey of CFI techniques proposes a quantitative security measure based on the average number of allowed targets per protected branch, along with the degree of the branch having the most allowed targets [19]. This is a challenging metric for any CFI technique based on static analysis because, for the program to run correctly, CFI must always allow *at least* the set of branch targets that the program needs to use. Many CFI tools resort to over-approximation where that set is difficult to determine. For BLACKBOX, the equivalent to this quantitative security metric is the number of targets that are trusted at each branch—the reliability of the IID log can be seen as the inverse of the number of trusted targets. But since an IID does not interfere with the monitored program, it has more leeway to raise concern about untrusted program behavior. Two case studies show another side to the Trusted Profiles from the G_{dnd} experiments, highlighting the narrow focus that can be achieved with IID profiling.

$G_{\mathbf{dbg}}$ Case Study: Untrusted Branches

Figure 3.6 shows that the vast majority of forward indirect branch sites in our G_{dnd} experiments are untrusted (dark lower section of each bar). At the same time, the rate of spurious anomalies was relatively low (Table 3.3), indicating that BLACKBOX profiling captured an accurate representation of normal application behavior. Even while using Outlook, Chrome, SciTE and pdflatex for everyday research and personal tasks, control flow throughout these highly complex programs held tightly to this stringent Trusted Profile. Where most CFI techniques strive to minimize the number of trusted targets at each indirect branch, BLACKBOX entirely distrusts the majority of indirect branches. This establishes a high probability of an anomaly report in the case that an installed CFI defense is compromised. It also gives developers high confidence that unusual program behavior will be reported for further investigation.

Figure 3.7 shows an even tighter Trusted Profile for the SPEC CPU 2006 benchmarks, but this is misleading from the standpoint of security evaluation. These applications are designed to minimize library calls, so it is not particularly meaningful that the majority of indirect branches are not used during the benchmarks. Figure 3.8 presents the same slice of the Trusted Profile for the main executable exclusively. The degree of security looks remarkably lower in this view, yet nothing has changed about BLACKBOX or the relative vulnerability of the monitored applications. While it is important to have common benchmarks that are used consistently throughout a particular research domain, it is also important to use the benchmarks in a meaningful way, and to carefully consider the kind of conclusions that can be reasonably drawn from the results.

For example, if the goal of the experiment is to demonstrate how effectively BLACKBOX monitors a single module, then focusing exclusively on the statistics of the Trusted Profile for the main module may be valid—unless the intended domain of monitored applications is very different from those main modules. Conversely, if the goal of the experiment is to demonstrate how BLACKBOX monitors an entire application, then these benchmarks are probably never valid, because they are designed to minimize library usage. In addition, the effectiveness of BLACKBOX monitoring is not just a factor of precision, but is also comprised of its false positive rate. If the goal of the experiment is to evaluate BLACKBOX for programs with a small, fixed set of inputs, then these benchmarks are representative of that scenario. But if BLACKBOX is intended for usage with desktop applications, where inputs are continuous and highly unpredictable at all layers of functionality, then these benchmarks cannot reasonably be used to evaluate the false positive rate in BLACKBOX.

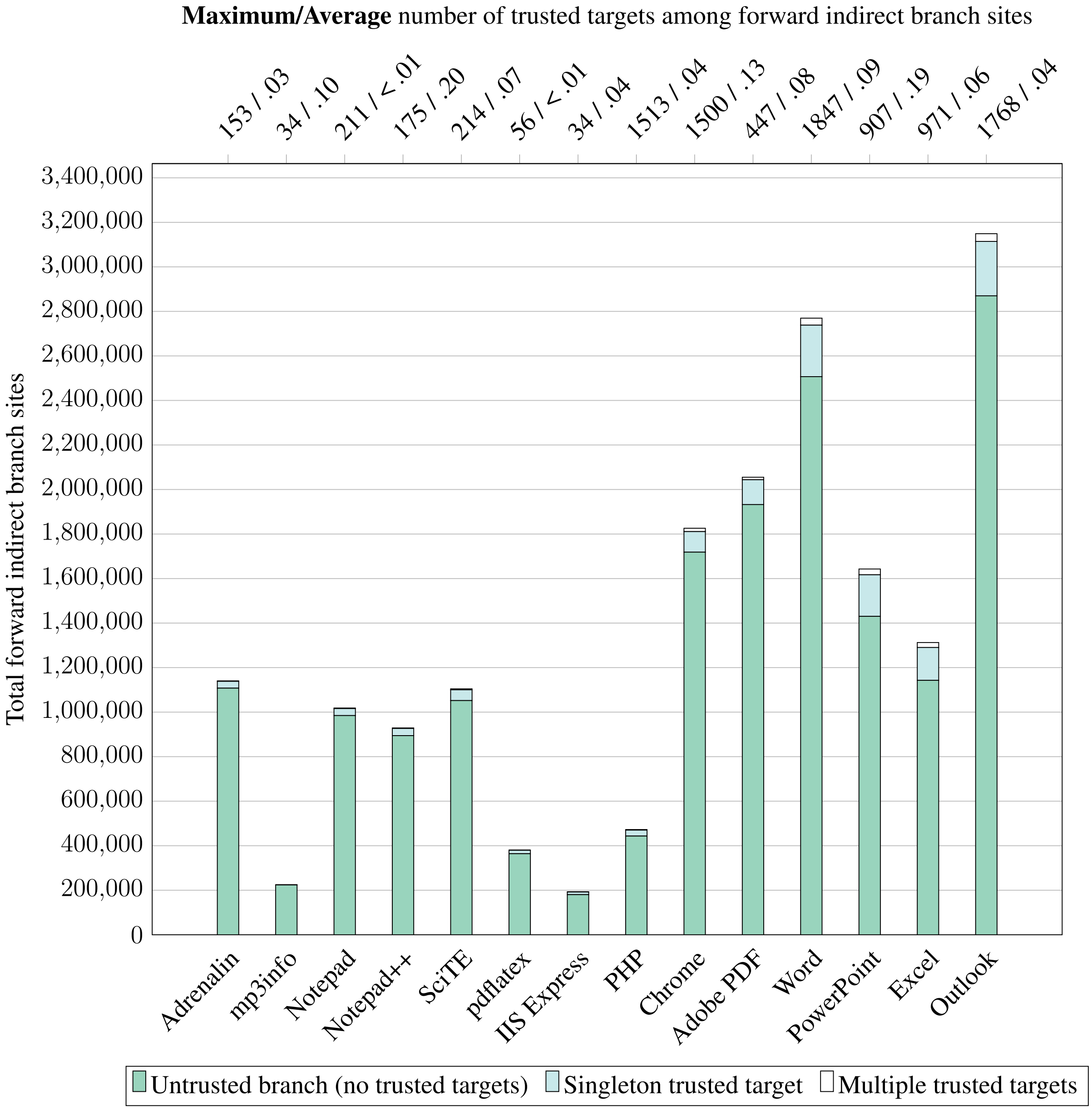


Figure 3.6: Summary of forward indirect branch sites in the Trusted Profile. Bars are split with untrusted branches in the large lower portion, followed by branches with a singleton trusted target, then branches where multiple targets are trusted. The maximum number of trusted targets as well as the average are listed across the top of the chart. The vast majority of indirect branches are entirely untrusted by BLACKBOX, such that any execution of the branch will result in an anomaly report.

Comparing CFI Approaches These factors make it difficult to compare BLACKBOX with CFI techniques in general, where the prototypes typically focus on server applications or the statically compiled components of web browser, which are highly regular applications and do not have the kind of characteristics that BLACKBOX excels at monitoring. Similarly, the relatively low code coverage of the SPEC CPU 2006 benchmarks also makes it complicated to compare results between static and dynamic CFI approaches. The static tool will usually report all indirect branches throughout each application because it actively secures them all, regardless of when or how they may be executed. Meanwhile the dynamic tool focuses on the small subset that is executed by the vendor-supplied input data, because it cannot secure a branch until the moment it is executed. So the fact of using the same benchmark does not necessarily establish a fair basis for comparison between two research evaluations.

$G_{\textbf{dbg}}$ Case Study: Cross-Module Branches

A particular limitation of static CFI is the cross-module branch. The difficulty arises from dynamically linked libraries, which are common in popular programs. Many CFI tools derive constraints from an offline static analysis, which makes these branches difficult to address because it cannot be known which particular modules (or which versions of modules) might be linked at runtime. But with seamless module support, BLACKBOX is able to incorporate these edges into the Trusted Profile, even where a single callback site targets private (i.e., non-exported) functions in multiple versions of the same library simultaneously (Section 3.3.4). Figure 3.9 depicts Trusted Profile statistics for cross-module targets from the previous G_{dnd} experiments, where each bar focuses on the number of cross-module branch sites in the corresponding Windows application. Bars are split according to the arity of trusted targets identified during BLACKBOX profiling for our experiments, with branch sites having only a singleton trusted target occupying the majority of each bar. Sniffer Dog is especially sensitive to the addition of a second target to a branch having only one trusted target, and will always assign the highest level of suspicion to such an anomaly.

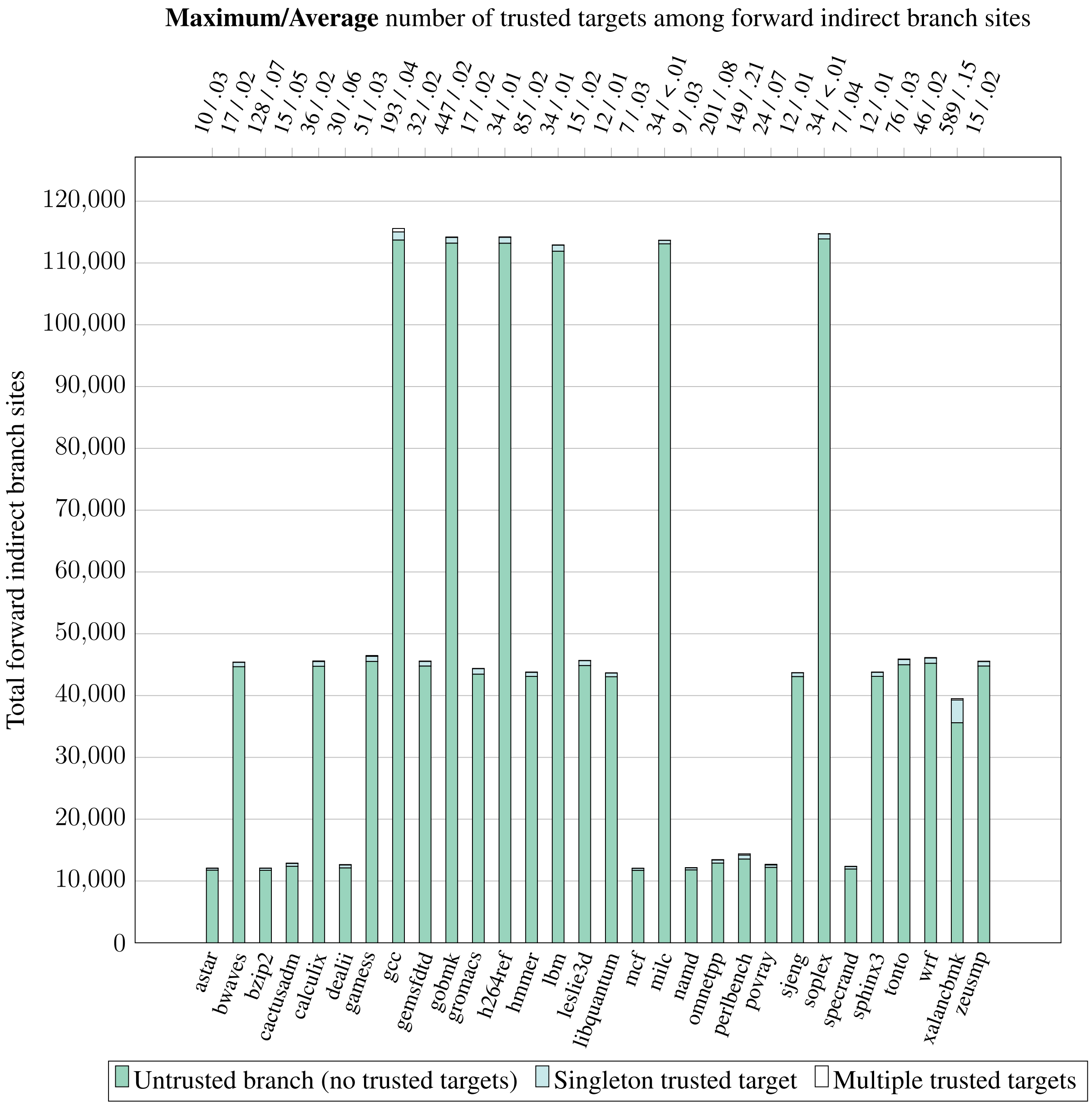


Figure 3.7: Summary of forward indirect branch sites in the Trusted Profile. Bars are split with untrusted branches in the large lower portion, followed by branches with a singleton trusted target, then branches where multiple targets are trusted. The maximum number of trusted targets as well as the average are listed across the top of the chart. The vast majority of indirect branches are entirely untrusted by BLACKBOX, such that any execution of the branch will result in an anomaly report.

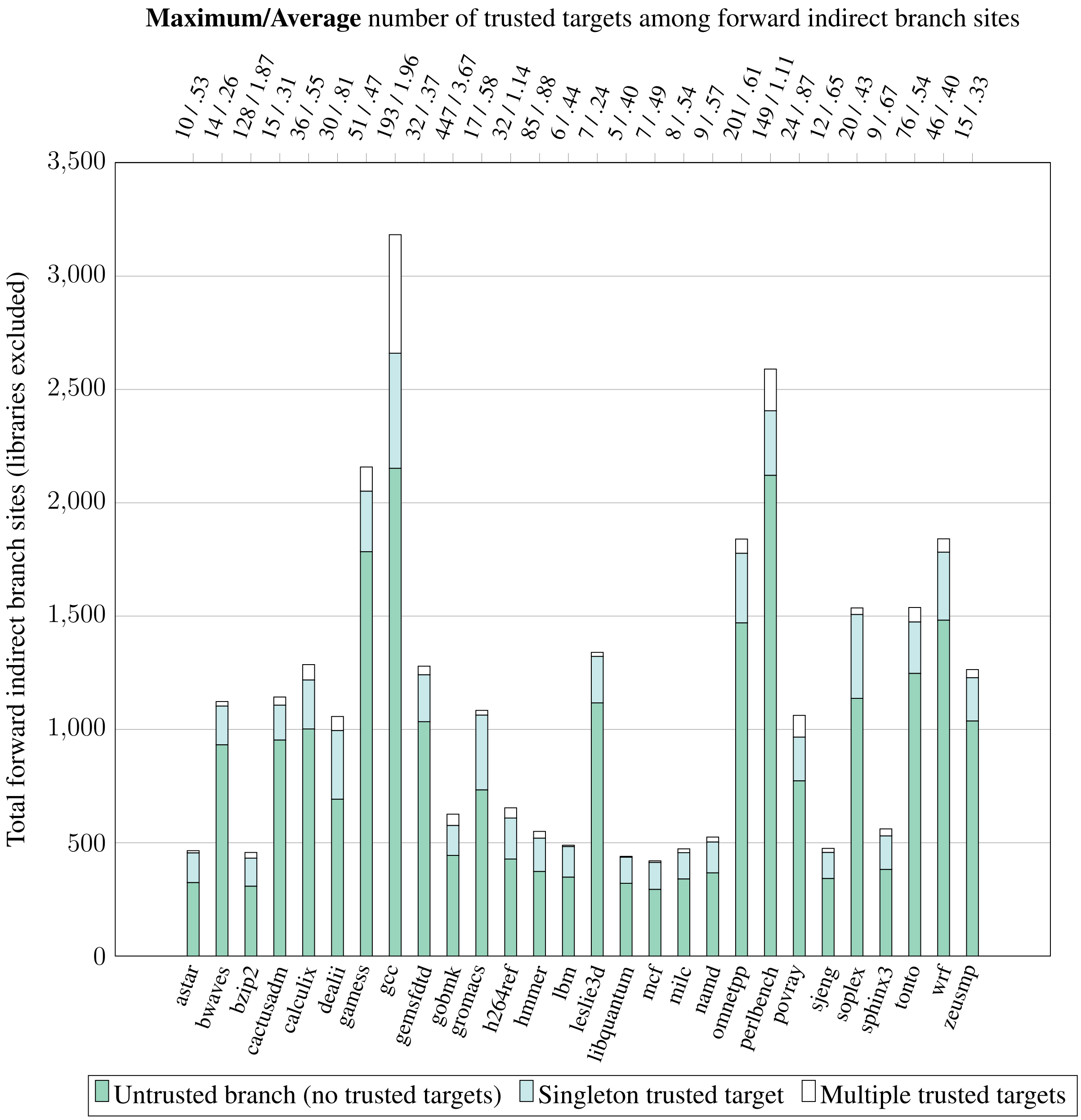


Figure 3.8: Summary of forward indirect branch sites in the Trusted Profile, excluding libraries. Bars are split with untrusted branches in the large lower portion, followed by branches with a singleton trusted target, then branches where multiple targets are trusted. The maximum number of trusted targets as well as the average are listed across the top. Even within the main module of a benchmark having inputs designed for code coverage, the majority of indirect branches are untrusted. Benchmark application `xalancbmk` has much higher indirect branch count and is omitted for uniformity of scale; its untrusted and singleton branch ratios are similar.

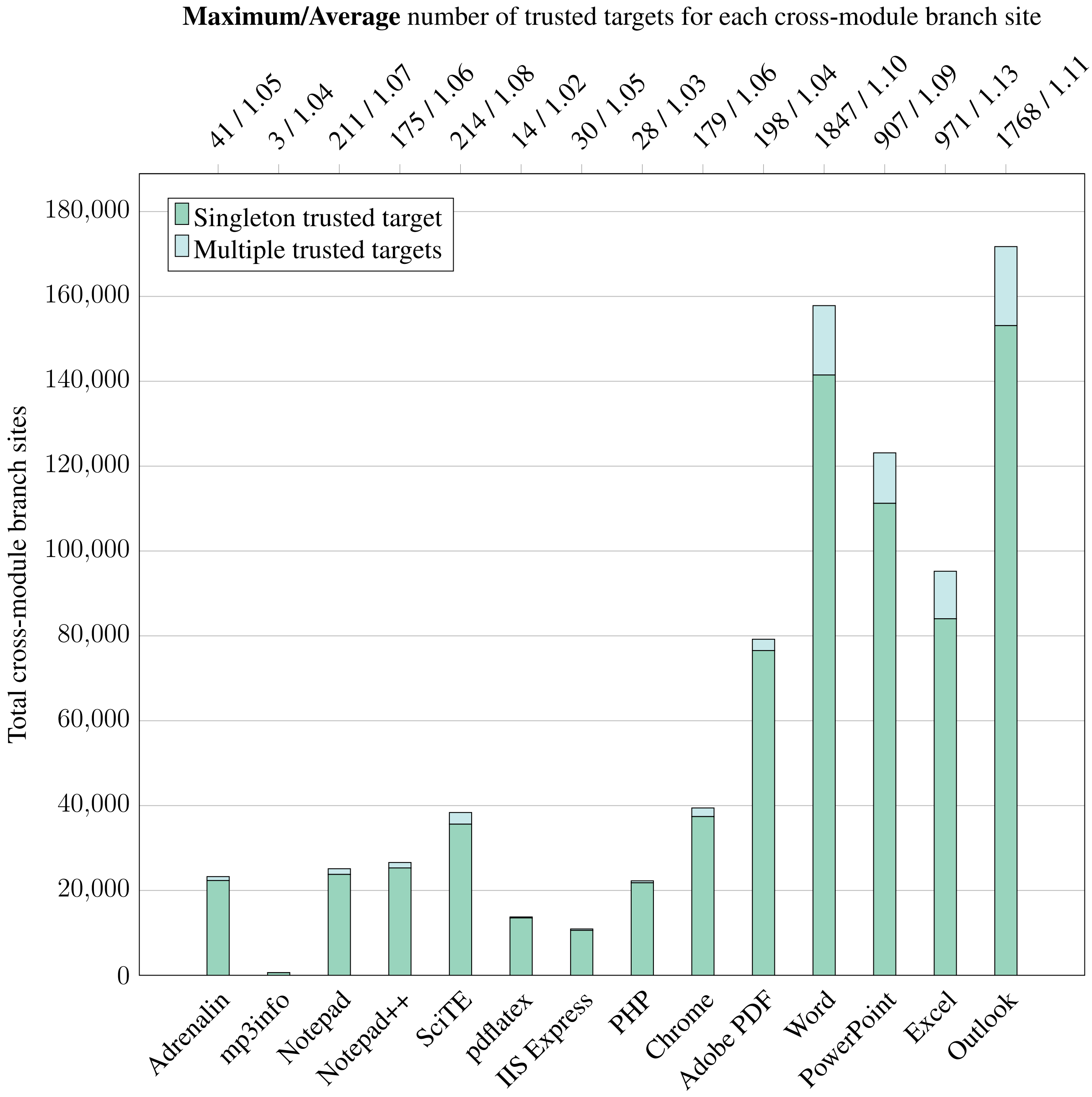


Figure 3.9: Summary of cross-module branch sites in the Trusted Profile. Bars are split by the arity of trusted targets, with branch sites having a Singleton trusted target at the base and Multiple trusted targets above. The maximum number of trusted targets as well as the average are listed across the top of the chart. The vast majority of cross-module branches only have one trusted target—and no branch site has more than 2,000 trusted targets—yet CFI based on module-unaware static analysis would permit these branches to reach any valid indirect branch target in the entire application.

Important CFI techniques proposed in research publications can be grouped into three categories according to their effectiveness in protecting cross-module edges. These categories will be outlined here in general terms, and revisited in more detail focusing on each CFI defense in Section 6.1.

1. **Accurate constraints** are applied by 3 kinds of CFI techniques, each having a distinct limitation in real-world deployment. The precision ranges from a single valid target per invocation (perfect constraints) to roughly 50% slack vs. a perfect static analysis.

    - Techniques that rely on the Last Branch Record (LBR) register have perfect accuracy for cross-module branches, just like every other branch—but other limitations make these approaches easily exploited (or the threat model explicitly declines to address activity occurring in other modules).
    - Purely dynamic approaches that protect vulnerable pointers at the transition from register to memory also have perfect accuracy for these branches. But these approaches tend to be slow in practice, and can be defeated by dynamic poisoning tactics that "borrow" security tokens for later reuse under certain conditions.
    - Monolith defenses that are limited to statically linked applications can apply their usual constraints, which are accurate up to the sophistication of the chosen intra-module static analysis. This includes defenses for Apple iOS and *nix kernels.

2. **Categorical constraints** are applied by many coarse-grained CFI approaches that enforce roughly the same accuracy as for intra-modular branches under the same CFI.

3. **No constraints** can be applied by `vtable` protection tools that do not offer runtime support for dynamically linked modules, although the presence of such tools may indirectly reduce this attack surface, since the dynamically supplied callback target is often a C++ method.

In summary, existing defenses either (a) constrain these branches to a broad category of targets, (b) neglect these branches entirely, or (c) expose unrelated weaknesses that a sophisticated attacker can reliably compromise. A cross-module branch attack will need to be much more selective to escape the attention of BLACKBOX, which in the vast majority of cases trusts exactly one target at each branch. Even the worst case branch allows just 1,847 trusted targets, making it orders of magnitude more responsive than coarse-grained CFI—though it is difficult to calculate the specific number of targets that a given CFI defense would allow for this particular branch, a reliable lower bound is the total number of locations that are ever reached by any indirect branch in our Trusted Profile. In comparison to the 1,847 trusted targets for this worst-case branch, any CFI lacking support for dynamic modules will allow it to reach at least 84,907 locations. The best case among coarse-grained CFI approaches will distinguish potential targets by function signature, reducing the set of allowed targets to a smaller category than its peers.

Figure 3.10 additionally indicates the number of cross-module callbacks that were trusted during profiling for the G_{dnd} experiments. These branches represent a worst case for existing CFI approaches because the callback targets are passed as function pointers. One CFI approach claims to provide accurate constraints on these branches, but a careful examination of its policy reveals that the allowed target set is the same as for any CFI based on static analysis alone. Only a fully dynamic CFI can protect these branches, and as mentioned above, these approaches have so far been found insecure for orthogonal reasons. In contrast, the Trusted Profiles from our experiments contain no more than 1.90 targets per callback site, with the worst case being the same branch in `wwlib.dll` of Microsoft Office having 1,847 targets under our usage. In summary, although many advances have been made in the effectiveness of CFI defenses, the quantity and severity of remaining vulnerabilities suggests the need for an intrusion detector like BLACKBOX.

Securing SPEC CPU 2006 can be a useful academic exercise, but does not necessarily represent the challenges faced by CFI when deployed on real-world applications. Figure 3.11 shows parallel statistics to Figure 3.9, leaving little doubt that cross-module edges are a much more significant vulnerability in Windows desktop applications than in these small command-line benchmarks.

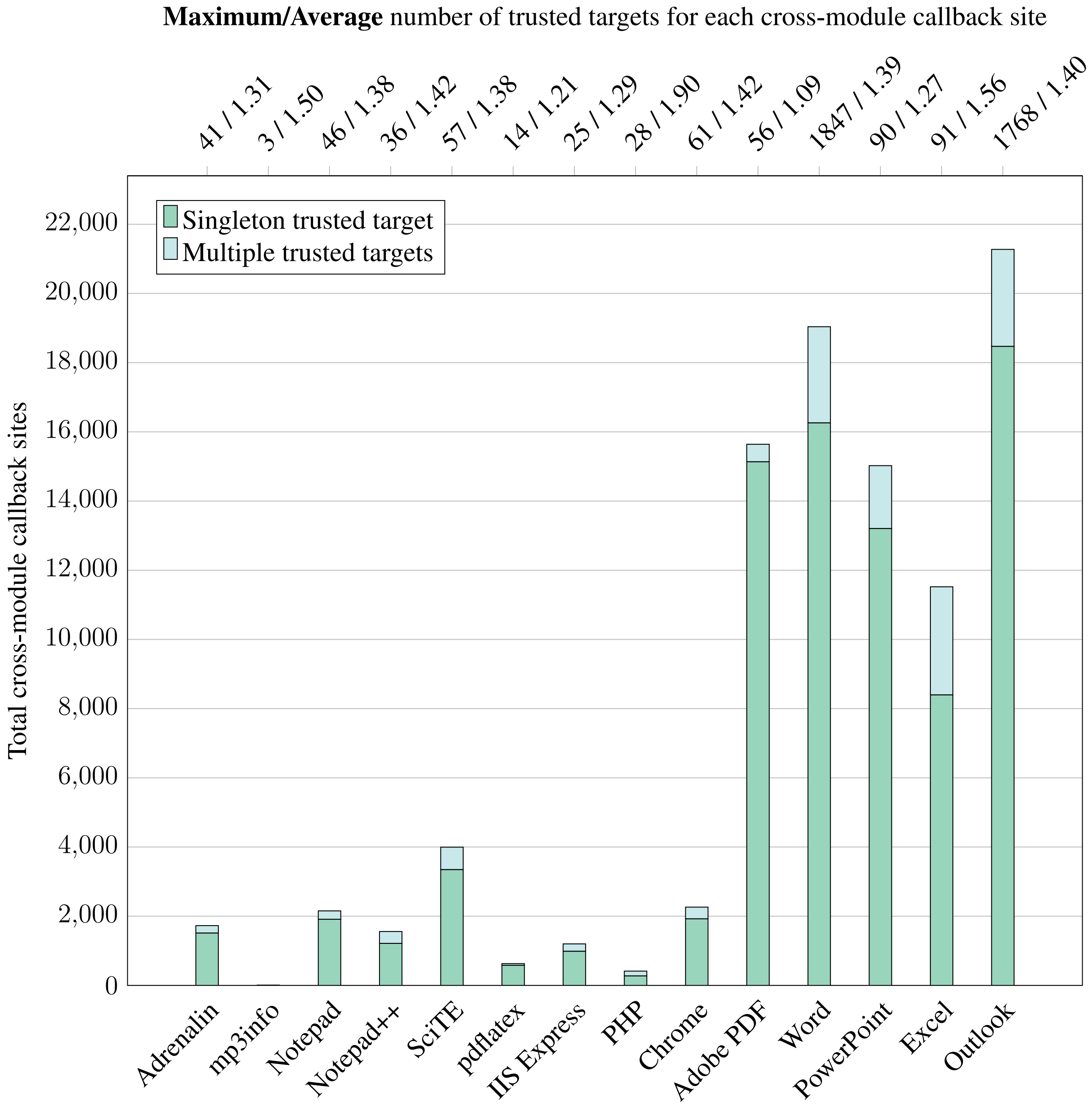


Figure 3.10: Summary of cross-module callback sites in the Trusted Profile (with callback aliasing disabled, which is less flexible for module upgrades—see Section 3.3.4). Bars are split by the arity of trusted targets, with callback sites having a Singleton trusted target at the base and Multiple trusted targets above. The maximum number of trusted targets as well as the average are listed across the top of the chart. A large percentage of cross-module callbacks only have one trusted target—and no callback site has more than 2,000 trusted targets—yet even module-aware CFI often permits these callbacks to reach any valid indirect branch target in the entire application.

Figure 3.12 shows the same parallel vs. Figure 3.10, again revealing an even smaller attack surface in this purely academic venue. While the reported results for CFI techniques may appear impressive, it is important to consider the consequences of deploying them on commercial software without having a reliable intrusion detector to fall back on when it becomes necessary.

3.6.5 Verifiability

BLACKBOX is open-source and can be used to repeat our experiments. The implementation was submitted to the CGO Artifact Committee and passed, indicating that the published results could be obtained by an independent third party. The versions of our applications reported in the experiments may not be available, especially those like Google Chrome that auto-update, but since BLACKBOX has no application-specific functionality, it should perform equally well on today's latest version of these applications.

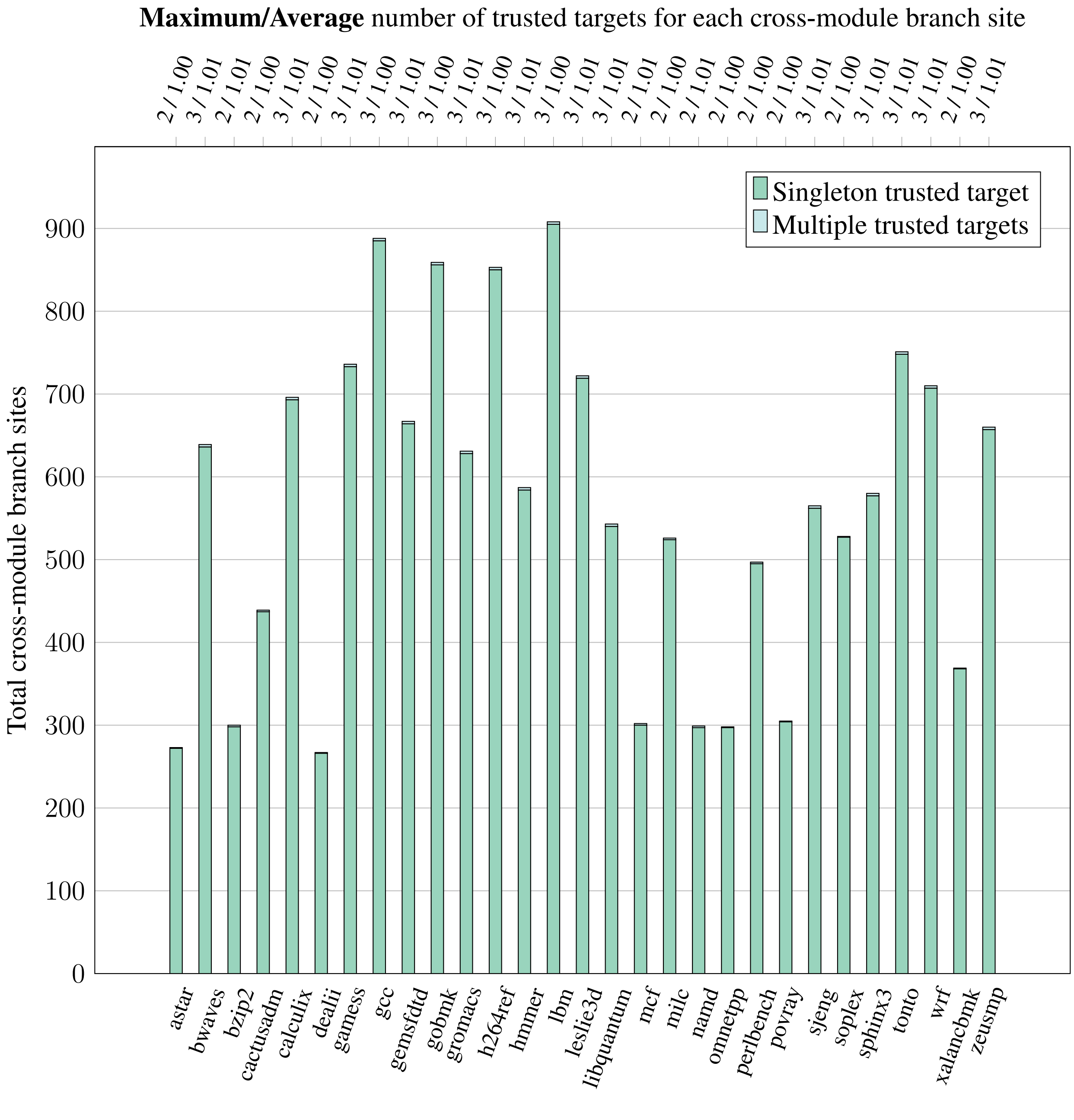


Figure 3.11: Summary of cross-module branch sites in the Trusted Profile of the SPEC CPU 2006 benchmarks (excluding libraries). The construction of these applications is not representative of typical end-user software that is commonly exploited today. For example, challenges to CFI techniques such as these cross-module branches occur at much lower frequency and with very few multi-target sites.

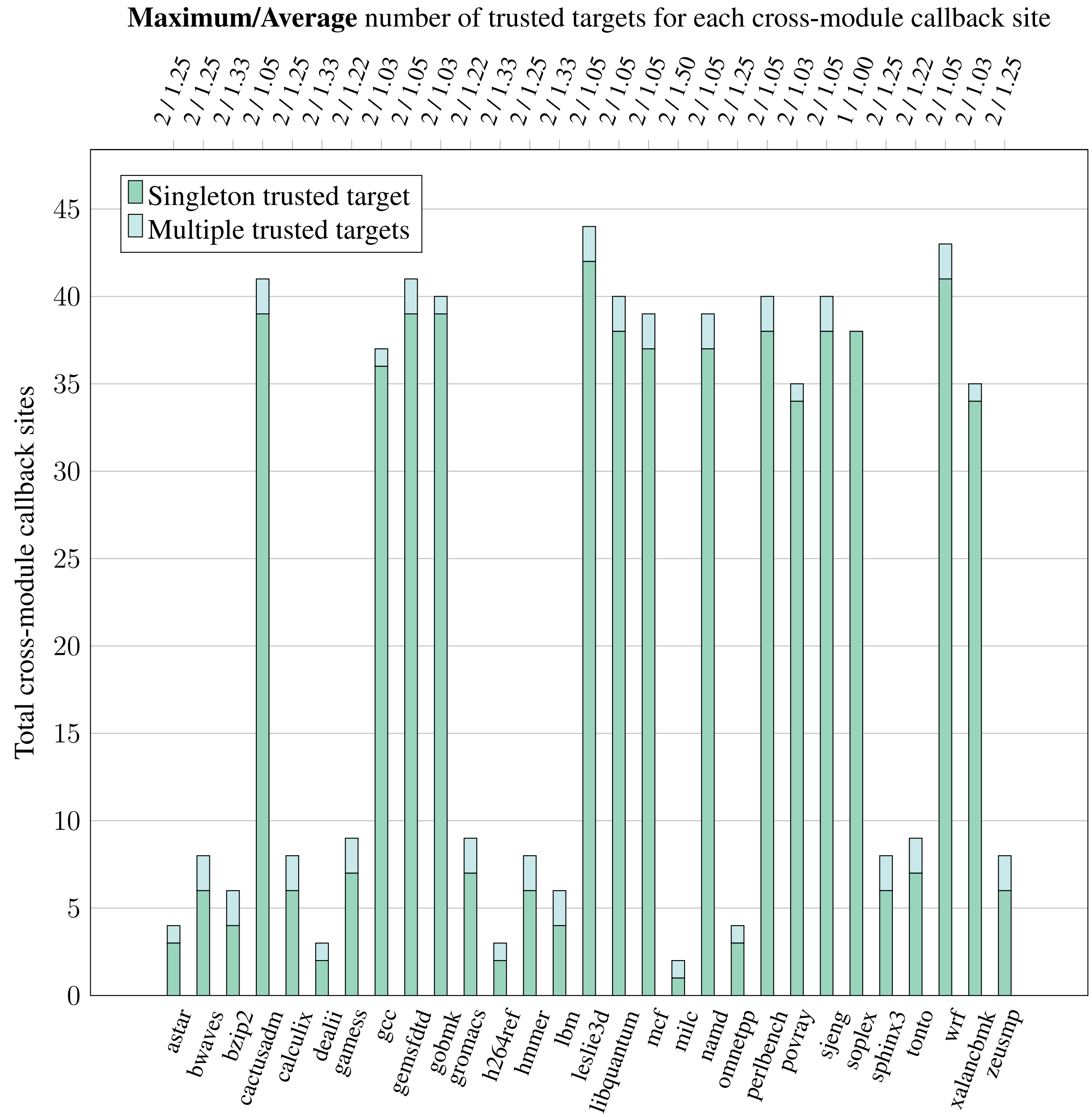


Figure 3.12: Summary of cross-module callback sites in the Trusted Profile of the SPEC CPU 2006 benchmarks (excluding libraries). The construction of these applications is not representative of typical end-user software that is commonly exploited today. For example, challenges to CFI techniques such as these callbacks occur at much lower frequency.

Chapter 4

DynamoRIO JIT Optimization

The performance goals of BLACKBOX are not feasible under current implementations of dynamic binary translation (DBT), because DGC presents a special challenge for the binary translation infrastructure, and current implementations do not resolve it efficiently. To maintain consistency between the original application and its translation in the code cache, a DBT tool must (1) detect modifications to the original generated code and (2) reconstruct the corresponding portion of the translated application. This can create tremendous additional overhead, reducing today's DBT platform performance by an order of magnitude for applications that frequently generate substantial amounts of code. As a baseline for comparison, one of the best cases for DBT is the SPEC CPU 2006 benchmark suite [168], where DynamoRIO [15] on 64-bit Linux averages merely 12% (Section 4.5). In contrast, the Octane JavaScript benchmark suite runs 9× slower in the Mozilla Ion JIT and 15× slower in the Chrome V8 JIT on the same platform. Similarly, the average overhead of Pin [105] is 21% [18] for SPEC CPU 2006, but Octane runs 6× slower in Ion and 10× slower in V8.

This is not just a problem for BLACKBOX. One of the major selling points of DBT over other dynamic analysis frameworks is that it can minimize runtime overhead while providing accurate and flexible introspection and instrumentation of the target application. Many optimizations have been developed, including translating the target application into an internal code cache such that each fragment of the translated application is only generated once—the first time it is executed. DBT tools also commonly implement advanced optimizations such as tracing and in-cache resolution of indirect branches. But these enhancements only apply to DBT performance for applications comprised exclusively of statically generated code—i.e., compiled binary executables and libraries. The slowdown for DGC substantially limits the viability of DBT for dynamic analysis use cases that involve frequent DGC, and so far this limitation has not been addressed.

There are many important use cases for DBT on applications having significant DGC. Security tools such as BLACKBOX and several of its predecessors [94, 165, 27] as well as bug detection tools [17] are especially important for DGC-intensive programs. In the first place, DGC is becoming more prevalent throughout today's popular applications. For example, all of the Microsoft Office applications use the JScript9 JavaScript engine to render built-in web browser components. Another common example is the Adobe PDF Reader, which renders the application frame in a Flash component, and implements built-in cloud account browsers in both Flash and JavaScript. Microsoft and Google host popular office suites online, where the majority of application functionality executes in the web browser's JavaScript engine. Since these JIT-intensive applications and components are most commonly used to access Internet resources, every security vulnerability becomes a publicly accessible attack surface, and every bug becomes a security vulnerability. Furthermore, the dynamic nature of generated code makes it vulnerable to a wider range of attacks, since some effective security measures such as $W \bigoplus X$ [110] are not compatible with the process of dynamically generating code.

Program analysis applications of DBT are similarly important for DGC-intensive programs because:

1. a wide range of popular and important software systems now include large DGC components;
2. the complexity of debugging DGC engines is much higher than for statically compiled code;
3. the majority of conventional analysis tools operate on source code, which for DGC only exists in the abstract form of internal data structures; and
4. performance is the primary goal of most dynamic code generators, making them ideal targets for the deep profiling and memory analysis that DBT excels at.

For DBT to be an effective platform for program analysis, it must be efficient enough for regular use, but this is not possible for DGC-intensive programs, given the current approach to translation consistency in popular DBT platforms.

Since this is an important problem across the entire domain of DBT, and is also essential for the viability of BLACKBOX, this chapter momentarily steps away from the subject of intrusion detection and focuses directly on optimizations for binary translation of dynamically generated code. Our augmented DynamoRIO significantly outperforms the state-of-the-art DBT systems on JIT programs. For the Octane JavaScript benchmark running in the Mozilla JavaScript engine, we achieve 2× speedup over DynamoRIO and 3.7× speedup over Pin, and for Octane in the Chrome V8 JavaScript engine we achieve 6.3× speedup over DynamoRIO and 7.3× speedup over Pin.

Optimization Techniques

The main reason for the extreme slowdown caused by DGC is that the DBT tool cannot easily detect when and where generated code is modified. While most RISC architectures require an explicit instruction cache flush request by the application to correctly execute modified code [90], this chapter focuses on the IA-32 and AMD64 platforms where the hardware keeps the instruction and data caches consistent and no explicit action from the application is required. Special measures must be taken by the DBT tool to detect code changes, as any memory write could potentially modify code. Since performance constraints do not allow instrumentation of every write in the entire application, the common approach is to artificially set all executable pages to read-only, and invalidate *all* code translated from a page when a fault occurs. These factors greatly increase the overhead of the DBT tool when an application frequently writes to pages containing code.

This chapter introduces two optimization approaches and demonstrates in the context of DynamoRIO that both can significantly improve performance of DBT for the JavaScript JIT engines V8 and Ion, achieving under 3× the native execution time on average. The simpler of these two approaches is to augment the target application with special source code annotations that are compiled into the binary and subsequently translated into DBT actions by the DBT interpreter. The specific annotations used are described in Section 4.2.1.

While the annotations have a very small impact on the native performance of the target application, the obvious disadvantage to this approach is that it requires source code and a special build of the target application. In addition, many applications are not trivial to annotate correctly, and annotation errors have the potential to cause the application to behave incorrectly or crash under DBT.

The second approach infers JIT code regions and instruments all writes targeting those regions to use a parallel memory mapping that has writable permission. The instrumentation also flushes fragments of the translated application that are invalidated by the JIT write. While this approach is less intrusive in the user's toolchain, it requires the use of additional address space and is more complex to implement.

Organization

The remainder of this chapter is structured as follows: Section 4.1 describes the basic structure of a binary translator in the context of DynamoRIO, and outlines the DGC strategies of four popular DBT platforms. Section 4.2 presents our new annotations targeting JIT optimization, followed by the annotation implementation details in Section 4.3. Section 4.4 then presents the JIT inference approach and Section 4.5 reports performance for each optimization.

4.1 Background and Related Work

The structure of a Dynamic Binary Translator is optimized for its performance on statically compiled code, which comprises the vast majority of target applications. This structure makes its performance especially weak for dynamically generated code. In its most naïve form, dynamic binary translation can be implemented as a pure interpreter, applying instrumentation each time a relevant program point is encountered. This approach would have identical overhead for both static and dynamic code. To optimize for the common case, DBT platforms typically translate fragments of the target application on demand into a memory-resident cache. This section begins with an implementation overview of DynamoRIO [15], highlighting the optimization details that benefit the common case but work against dynamically generated code. The remainder of the section outlines the strategies for handling DGC used by popular DBT platforms.

4.1.1 DynamoRIO

DynamoRIO initially translates each basic block of the target application on demand into the code cache, linking the translated blocks in parallel with their original counterparts to replicate the original control flow within the cache. As new blocks of the target application are executed, the code cache is incrementally populated until eventually the application runs entirely within the cached copy. An indirect branch in the control flow may have many targets, which are specified by an address value in the memory space of the application. Since the data flow of the target application is identical to a native run, the address of the branch target always refers to the original application's memory—untranslated code outside the code cache. To prevent the execution from returning to the original application, the branch is redirected to a lookup routine within the cache that finds the translated code fragment corresponding to the branch target and jumps to it. Figure 4.1 depicts the translation and linking process.

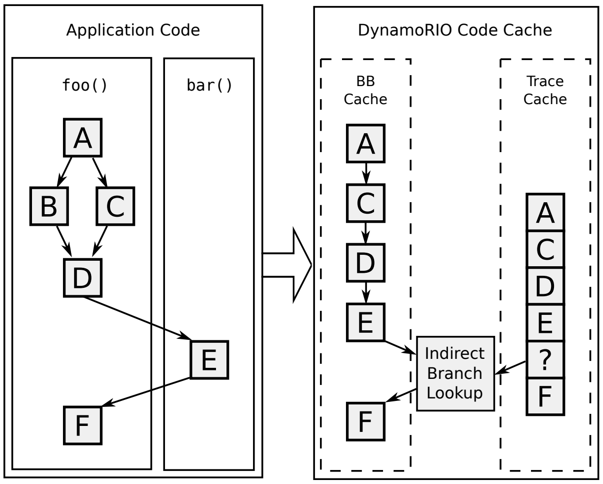


Figure 4.1: Overview of Dynamic Binary Translation. Blocks of application code are dynamically translated into a code cache where they are linked back together.

Traces in DynamoRIO To improve performance for hot paths in the application, DynamoRIO instruments each indirect branch target with an execution counter, and when the count reaches a configurable threshold (by default 52), the fragment is designated as the head of a trace. As execution continues, the trace is progressively extended using the Next Executing Tail scheme. Indirect branches within a trace are translated into the more efficient direct branches, which do not require a lookup routine, and direct branches are removed by fusing the basic blocks of the trace into a single straight-line code sequence.

VM Areas in DynamoRIO When the target application allocates memory, whether by `mmap` to load a module image or by an allocation function such as `malloc` or any other means, DynamoRIO creates an internal accounting structure for that block of memory called a VM area. Executable code in a VM area is associated with it by a list of code fragments, which can be either basic blocks or traces. When a module image is unloaded by the target application, the corresponding VM area with its list of fragments is flushed accordingly. Likewise, if the application removes executable permission from a region of memory, the corresponding code fragments must be flushed—even if the area becomes executable again, changes to the code will not be known, and any obsolete fragments will cause the application to behave incorrectly. To detect changes to code in a VM area that is both writable and executable, DynamoRIO artificially sets the page permission to read-only [16]. When a page fault occurs in the VM area, all of its code fragments are flushed. While it is possible to identify the specific fragments that were changed and flush them selectively, it would require special handling of memory permissions to execute the write exclusively of other threads, because even a momentary change to the writable permission is global and could allow a concurrent thread to write undetected. This problem is one of the root factors that makes naïve handling of DGC inefficient, so we refer to it throughout this paper as the *concurrent writer problem*. In general, the minimum granularity of a dynamic code change is one page of memory, because a write cannot be executed exclusively within a virtual page. The optimization in Section 4.4 introduces an alternative, more sophisticated approach that eliminates the *concurrent writer problem* and enables finer granularity code changes.

DGC in DynamoRIO While this structure is very efficient for module images loaded via `mmap`, it requires most dynamically generated code fragments to be translated into the code cache many times repeatedly. Consider the simple case of a JIT engine generating a compiled function, executing the function, then generating a second function in the same VM area and executing it, and so on. The writing of each new function requires all the existing functions to be invalidated and retranslated. This process is especially cumbersome for traces, which must be rebuilt after each VM area flush according to the hottest paths identified by the instrumented trace heads. While it is possible to simply disable tracing for the JIT code areas, this yields very poor performance in long-running benchmarks such as Octane. Since the JIT compiles the hottest paths in the application's JavaScript, the JIT code areas are necessarily the hottest paths in the application. Disabling traces improves the efficiency of the code generation process, but in a full run of Octane costs 25% in overall execution time because the code along the hot paths is so much less efficient in CPU execution time.

The benchmark results in Section 4.5 indicate that DBT in general performs much worse on Chrome's V8 JavaScript engine than on Mozilla's Ion, even though V8 outperforms Ion in a native run. This is caused by frequent writes to small data areas, typically bit fields, scattered throughout the generated code. Since these writes do not affect translated code, DynamoRIO could theoretically execute the writes without flushing the region. But this is prevented by the risk of a concurrent write—and even if it were possible, it would still be very expensive to determine whether the write target overlaps any translated code fragment. The search would be reasonably efficient if DynamoRIO were to keep a sorted data structure of spans, but it does not because that large and expensive structure would be of no value for module image VM areas, which comprise the vast majority of translated code. Therefore, without special optimization for DGC, determining fragment overlap would require a time-consuming search of the code fragment list for the targeted VM area.

4.1.2 QEMU

When QEMU [10] translates code fragments from a page of guest memory, two strategies can be used to detect code changes. The first strategy is similar to DynamoRIO, marking the page read-only and handling the fault as if it were a write event. The second strategy relies on the QEMU softmmu layer, which provides a software TLB that effectively maps the guest page table to the host page table. When the guest writes to a page of memory, the target is translated through the softmmu layer to the corresponding host page, which can trap into QEMU for code change handling [108].

4.1.3 Pin

Pin [105] translates all code into traces. For any trace containing instructions that were dynamically generated, the head of the trace is instrumented to check whether any of those instructions have changed [116]. When executable permission is removed from a page containing DGC, all traces containing code fragments translated from the page are invalidated. During periods of frequent code generation, this approach is more efficient than instrumenting every store, because traces will be executed much less frequently than stores. But the cost increases dramatically while the JIT engine is dormant and the generated traces are repeatedly executed, since the instrumented checks rarely discover code changes and are executed far more often than stores (assuming the generated code is collectively hotter than the interpreted code, and that the generated code does not itself generate code). Both DynamoRIO and QEMU rely on detecting code changes at the time of the write, leading to the *concurrent writer problem*, but the Pin approach relies on detecting code changes at the time the translated traces are executed. This makes it possible for Pin to selectively flush individual traces from the code cache. The benchmark results in Section 4.5 show that this approach outperforms region flushing for executions of the Octane benchmark.

4.1.4 Valgrind

Valgrind [120] provides two methods for synchronizing its code cache with dynamically generated code. The first is similar to Pin, instrumenting every dynamically generated basic block with a check for modified code. The second strategy requires compiling the target application with a source code annotation [156] that is translated into a code cache flush event. While the latter approach is relatively efficient, it does not significantly improve performance because the cost of both methods is overwhelmed by the slowdown of Valgrind's translation of basic blocks through a three-value IR.

DynamoRIO implements some of the Valgrind annotations for compatibility purposes. In addition, Section 4.3 presents a new annotation scheme that is much more efficient than Valgrind's. It also supports all popular compilers on the Windows 32-bit and 64-bit platforms, including Microsoft Visual Studio, the Intel C++ Compiler and GCC.

4.1.5 Specialized Applications of Binary Translation

The **Transmeta** Code Morphing Software$^{\text{TM}}$ [44] leverages hardware support to synchronize with dynamic code in several ways, including: (1) an approach like DynamoRIO's, but with sub-page write detection, (2) the technique now used by Pin, (3) a similar approach which only revalidates a DGC fragment after its containing write-protected region has been written, (4) patching translated code on the basis of recognized DGC patterns such as jump target substitution, and (5) translating frequently modified code into *translation groups*, which cache a history of recent translations for recurring DGC.

Librando [77] automatically diversifies the output of a JIT compiler at runtime for increased security. Librando allows the JIT to run natively, and detects JIT writes using the same page protection scheme as DynamoRIO, additionally minimizing overhead by validating each diversified basic block using a hashcode of the instruction bytes.

A survey by **Keppel** [91] outlines a variety of techniques for detecting self-modifying code in instruction-set simulation.

4.2 Annotation-Based Optimization

A simple approach to optimizing DynamoRIO for JIT code is to add source code annotations to the target application that notify the DBT about changes in generated code. Section 4.3 describes the design and implementation of our underlying annotation scheme. This section introduce ANNOTATIONDR, an extension of ORIGINALDR that supports specific event annotations for JIT optimization.

4.2.1 New Annotations Identifying Code Changes

In ANNOTATIONDR, we introduce three annotations to facilitate handling DGC in DBT systems:

1. `ManageCodeArea(address, size)`: disables the default method of detecting code changes for the specified region of memory until the area is unmanaged.
2. `UnmanageCodeArea(address, size)`: re-enables the default method of detecting code changes for the specified region of memory.
3. `FlushFragments(address, size)`: flush all fragments of the translated application corresponding to the specified region of the target application.

These annotations indicate the allocation and deallocation of JIT code regions and notify ANNOTATIONDR of all writes to the JIT code VM areas. We refer to JIT code regions as *managed code regions*. The default write detection schemes are disabled for managed code regions in ANNOTATIONDR, because every change to JIT code is explicitly annotated.

Given a correctly annotated application, the remaining challenge is minimizing redundant code fragment flushes by providing a finer granularity flush operation. ANNOTATIONDR introduces two improvements, described below.

4.2.2 VM Area Isolation

Recall that by default, ORIGINALDR invalidates cache code at the granularity of VM areas. A substantial performance improvement can be made in ANNOTATIONDR's handling of DGC by simply reducing the size of the VM areas containing DGC to single pages. Annotations indicate which memory allocations of the target application contain JIT code, so the corresponding VM areas can either be split into single pages upon instantiation, or lazily as code in those regions is invalidated. The annotation event for JIT writes allows ANNOTATIONDR to avoid the expensive page faults required by the default strategy for detecting dynamic code changes, and more importantly alleviates the *concurrent writer problem*. Together these improvements reduce the execution time to 3.7× native execution time on the Octane JavaScript benchmark [23] for V8 [66] and 2.6× for Ion [117] (see Section 4.5).

4.2.3 Selective Fragment Removal

Further improvement requires identifying, for each JIT write, which specific code fragments should be invalidated and selectively removing them from the code cache. For small writes such as a jump target change, it is alternatively possible to patch the corresponding fragments in the code cache instead of removing them. However, this would add significant complexity to the interface used to build tools with ANNOTATIONDR. Tool authors would need to spend effort handling the case of instrumented code being directly patched, including jump targets changing, which can drastically affect security or sandboxing tools. By instead simply invalidating fragments corresponding to the modified code we keep the tool interface consistent. The tradeoff here between complexity and performance may be worthwhile for specific tools and could be explored in future work.

The invalidation process is made more complicated by several implementation factors of ORIGINALDR: (1) the constituent basic blocks of each trace are kept separately from the trace itself, (2) traces typically do not consist of code that was contiguous in the original binary, and (3) a single basic block in the target application may be translated into several overlapping basic blocks, each having a different entry point and all having the same exit. As discussed at the end of Section 4.1.1, this requires an efficient data structure for sorted spans. The conventional data structure would be a red-black interval tree, but given the large quantity of DGC fragments, each traversal would likely incur at least a dozen branch mispredictions, and probably several CPU cache misses as well. Instead, ANNOTATIONDR stores the DGC fragment spans in a hashtable in which each bucket represents 64 bytes of a JIT code VM area. When a DGC fragment is translated into the code cache, its span is added to each hashtable bucket it overlaps. To lookup fragments overlapping a JIT write, the hashtable is first consulted to obtain a list of buckets whose 64-byte span is touched by the write. Since it is common for JIT engines to intersperse the generated code with small data fields, ANNOTATIONDR next checks the fragments in each bucket for overlap, such that only fragments specifically overlapping the written bytes will be flushed. A single hashtable bucket can in rare cases contain up to 32 fragments (for example, in a field of two-byte trampolines), so to minimize CPU cache misses while traversing a bucket's chain, each link in the chain holds 3 fragment spans. Figure 4.2 illustrates the distribution of code fragments into the hashtable buckets.

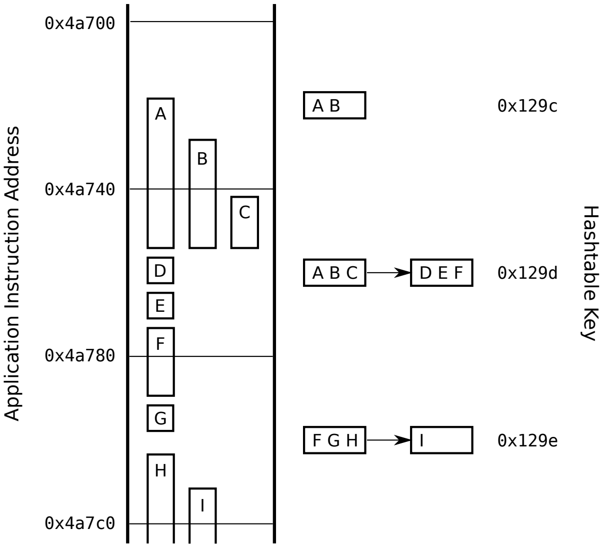


Figure 4.2: Distribution of translated code fragments into the overlap hashtable. To locate fragments overlapping a JIT code write, ANNOTATIONDR first looks up the buckets overlapping the span of bytes written by the JIT, then identifies the specific code fragments that were overwritten (if any).

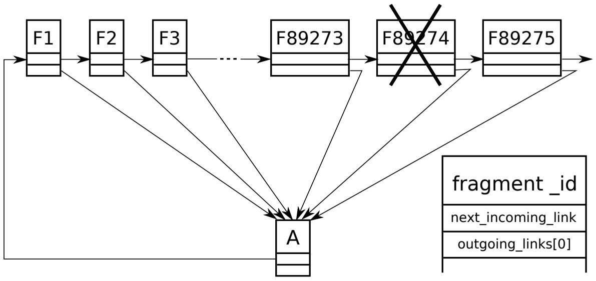


Figure 4.3: The singly-linked list of incoming branches requires an $O(n)$ traversal to remove a fragment, resulting in significant overhead for high fan-in basic blocks that are common in JIT code.

High Fan-in One challenge inadvertently raised by this optimization is that the removal of incoming direct branches can become a significant bottleneck. In statically compiled code, direct branches are rarely removed, so ORIGINALDR optimizes for space by using a singly-linked list to identify all direct branches targeting a code fragment (Figure 4.3). But in a run of Octane's Mandreel benchmark, in which V8 generates up to 150,000 direct branches that all target the same basic block, these branches are often removed during execution as stale code fragments are overwritten. The corresponding fragment removal requires an $O(n)$ traversal of a potentially long branch list, and in the case of Mandreel this consumes the entire speedup of selective fragment removal.

This slowdown can be alleviated by observing that the majority of high fan-in branches come from basic blocks that are no longer in use by the JIT, but have not been overwritten or deallocated (i.e., dangling fragments). For this reason, the slowdown does not occur in ANNOTATIONDR with VM area isolation, because the coarser flushing of whole pages randomly eliminates the majority of incoming branches from dangling fragments. ANNOTATIONDR simulates this effect by limiting the number of incoming direct branches to 4096, removing the fragment containing the oldest incoming branch when the limit would otherwise be exceeded. In some cases the removed fragments may still be in use, requiring retranslation, but in general the removed fragments are never executed again. This heuristic is not necessarily optimal for all JIT scenarios, so to handle an extreme case it is possible to selectively substitute the linked list with a hashtable of incoming direct branches.

4.3 Annotation Implementation

This section describes how our annotations are implemented. Beyond our primary use of annotations for identifying DGC, there are many other use cases for annotations in a DBT platform. Existing annotations are often for avoiding false positives in tools such as memory usage checkers. For example, Valgrind provides several dozen annotations with such functionality as printing to the Valgrind log, or marking variables in the target application as having defined values in cases where the Memcheck [156] tool's analysis would otherwise report an undefined-use error.

4.3.1 Binary Annotation Scheme

Source code annotations in the target application are compiled into binary annotations which can be translated by the DBT tool into event callbacks at runtime. A binary annotation is a short code sequence that has the effect of a `nop` during native executions of the application, but can be definitively recognized by the DBT interpreter. Annotations must support passing argument values to the DBT tool and returning a return value to the application. These argument values are constructed using normal compiled code.

There are three requirements for implementing effective annotations:

1. The annotation must create minimal overhead during native execution of the target application, such that annotations can be included in the release build.
2. Compiler optimizations can potentially modify or even remove annotations, so they must be defined in such a way that the DBT tool can recognize it after optimizations have transformed it.
3. The annotations must be distinctively recognizable by the DBT tool, such that annotation detection does not create overhead for code that does not have annotations.

```
401972 jmp    401985      # first jump
401974 mov    0x202688,%rax            # name base
40197c bsf    0xffffffffffffff98,%rax # name offset
401985 jmp    401996      # second jump
401987 mov    $0x2,%esi   # argument 2
40198c mov    $0x1,%edi   # argument 1
401991 callq  4024f2      # annotation function call
```

Listing 4.1: Annotation macro for 64-bit GCC. The first jump operand is always 0x11 bytes, allowing the DBT tool to use it as a quick detection filter. The subsequent `mov` and `bsf` encode a pointer to the annotation name in the text section.

Our basic approach to satisfying Requirement 1 is to prefix the annotation with a jump beyond the annotation, such that the body of the annotations is dead code—skipped over in a native execution (Listing 4.1). Implementing the annotation using inline assembly meets all three requirements, as (1) the jump over the annotation has near zero runtime overhead, (2) inline assembly instructions are never transformed by compiler optimizations (even whole-program optimizations), and (3) the exact byte sequence can be precisely controlled to minimize ambiguity with application code.

The developer invokes the annotation using a macro, which accepts ordinary arguments from the domain of application variables, constants and functions, e.g.:

```
MANAGE_CODE_AREA(start, compute_area_size());
```

In this example, even if the compiler inlines the entire function body of `compute_area_size()` into the argument preparation, it would all be skipped during native execution.

4.3.2 Annotation Discussion

Existing DBT tools usually use one of two annotation types:

1. a call to a function that is empty except for a short instruction sequence to prevent "identical code folding" optimizations from transforming all the annotation functions into a single function [155, 173], or
2. a sequence of instructions inserted at the annotation site that has the effect of a `nop` during native execution but can be distinctively recognized by the DBT tool.

The empty functions are simple and flexible, allowing any number of arguments to the annotation, though the cost of the empty function call increases with the number of arguments (Table 4.1). Valgrind's approach limits annotations to 5 arguments and requires more effort for the DBT tool to detect because the distinct instruction sequence is 12 bytes long for 32-bit applications and 16 bytes long for 64-bit applications. This not only requires more bytes to be examined during annotation detection, it also causes a complication when a basic block is truncated in the middle of an annotation. In this case, the DBT tool must either maintain a state flag for each thread indicating that the last decoded instruction could be part of an annotation, or to simply read beyond the end of a basic block to see if the subsequent bytes might form an annotation. The former approach is wasteful because the flag must be checked very frequently and it is almost always off, while the latter approach risks a segfault if the forward bytes are not readable. Valgrind's annotations also rely on GCC pragmas which are not available in other compilers.

Our annotation scheme using a jump over argument setup improves on both approaches. It is more efficient than always performing a function call. It also out-performs existing inline assembly

	One arg	Five args	Five varargs
ANNOTATIONDR annotation	1.54×	1.54×	1.54×
TSan annotation	2.35×	3.34×	4.01×
Valgrind annotation	3.05×	3.05×	N/A

Table 4.1: Native execution overhead of binary annotations in the extreme case of annotating every array index expression in the SPEC CPU 2006 benchmark 470.lbm.

annotations, as (1) the annotation is completely skipped by direct jumps during a native run (including argument setup code), (2) the annotation arguments use as many registers as the platform calling convention allows, and (3) the presence of an annotation can more efficiently detected. Table 4.1 shows the annotation overhead in an extreme case of adding 486 annotations in the inner loops of the `lbm` array computation from the SPEC 2006 benchmark suite. Even though the annotation is executed every time an array value is indexed, the overhead is just 50%. Detecting the annotation in a DBT tool still requires reading beyond the end of a basic block in case it has been truncated in the middle of an annotation. But our approach minimizes the cost of a potential segfault by minimizing the number of conditions in which (1) the risk is taken and (2) there is actually no annotation present. We selected the first two instructions of the annotation to be relatively rare for normal code: (1) a direct short jump of fixed length (which varies per platform), and (2) `int 2C`, which is the only instruction in the Intel x86 ISA to start with byte `CD`. A basic block will usually terminate at the jump, so we only need to read the next byte to determine whether further unsafe reading is required to complete the detection, and this one-byte read is relatively inexpensive because in most cases it lies on the same page as the jump, which guarantees it must be readable. Note that the approach is slightly different for 64-bit Windows where inline assembly may not be supported (Section 4.3.3).

4.3.3 Annotations in 64-bit Microsoft Visual Studio

The 64-bit Microsoft Visual Studio compiler presents a special case for the annotations because it does not support inline assembly. Without inline assembly, the annotation must be defined in ordinary C code, from which the compiler may produce a broad range of assembly sequences. Compiler optimizations such as dead code elimination make this especially complex because the annotation itself is dead code—in a native run the execution always jumps over the annotation body. It is also common for the compiler to move parts of the annotation, for example sharing a single register load between two annotations within the same function, which would make the annotation unrecognizable to ANNOTATIONDR because the register load is one of the key identifying elements of the annotation.

Listing 4.2 presents the C code for the ANNOTATION() macro, which takes the following arguments:

- `annotation`: the name of the annotation function, as defined in the target application.
- `native_code`: a block of statements to execute instead of the annotation during native execution of the application (may be empty).
- `...`: arguments to the annotation function, corresponding to the formal parameters of the annotation function.

The annotation begins and ends with conditional branches, both of which are never taken during native executions. The three values used in the conditional branches are specially selected to prevent compiler analyses from determining that the condition will always be false:

```
#define HEAD (0xfffffffffffffff1 - (2 * __LINE__))
#define TAIL (0xfffffffffffffff0 - (2 * __LINE__))
#define GET_RETURN_PTR() \
  ((unsigned __int64) __AddressOfReturnAddress())
#define ANNOTATION(annotation, native_code, ...) \
do { \
    if (GET_RETURN_PTR() > HEAD) { \
        extern const char *annotation##_label; \
        __int2c(); \
        _m_prefetchw(annotation##_label); \
        __debugbreak(); \
        annotation(__VA_ARGS__); \
    } else { \
        native_code; \
    } \
} while (GET_RETURN_PTR() > TAIL)
```

Listing 4.2: Annotation macro for 64-bit Microsoft Visual Studio.

- `GET_RETURN_PTR()` leverages intrinsic function `__AddressOfReturnAddress` to obtain a pointer to the return address on the stack.
- `HEAD` generates an integer larger than any stack address on 64-bit Windows (the built-in macro `__LINE__` is substituted with the source code line number on which the annotation macro is used).
- `TAIL` generates another such integer that is distinct from any HEAD in an annotation on any subsequent line in the source file. This prevents ambiguity between the tail of one annotation and the head of the next (even if multiple annotations appear on the same line).

The `__int2c()` serves as a hint to ANNOTATIONDR that the preceding branch belongs to an annotation. The first byte of this instruction is distinct from the first byte of any other instruction on x86-64, making it possible to examine just one byte following any direct branch and determine with high accuracy whether it could be part of an annotation. Since the compiler must regard an interrupt similar to a memory fence, the interrupt is guaranteed not to be reordered by compiler optimizations away from the branch instruction. The prefetch of `annotation##_label` is the unique identifier of the annotation, from which ANNOTATIONDR determines (1) that the instruction sequence is definitely an annotation, and (2) the name of the annotation function (since there may be no symbol associated with the annotation function call, for example in a stripped binary). The `__debugbreak()` instruction simplifies ANNOTATIONDR's parsing of the annotation by preventing the compiler from interleaving argument setup for the annotation call with the prefetch of the annotation label (since an interrupt is regarded similar to a memory fence). Note that while this definition of the annotation macro is robust in practice on all available versions of Microsoft Visual Studio, it is possible that future versions of the compiler may require the annotation macro definition to be revisited. Listing 4.3 shows a sample of an annotation compiled in Microsoft Visual Studio 2012.

```
; if (GET_RETURN_PTR() > HEAD)
1400019B6: lea  rax,[rsp+0C8h]
1400019BE: cmp  rax,0FFFFFFFFFFFFFDEFh
1400019C4: jbe  1400019E5
; annotation hint: byte CD follows the branch
1400019C6: int  2Ch
; annotation label (register or immediate operand)
1400019C8: mov  rax,<constant>
1400019CF: prefetchw  [rax]
; int3 isolates prefetch from argument setup
1400019D2: int  3
; argument setup code
1400019D3: mov  edx,1
1400019D8: mov  ecx,[140030160h]
; call annotation_log()
1400019DE: call annotation_log
1400019E3: jmp  1400019F7
; native_code: calls printf()
1400019E5: mov  edx,[140030160h]
1400019EB: lea  rcx,<constant>
1400019F2: call printf
; while (GET_RETURN_PTR() > TAIL)
1400019F7: lea  rax,[rsp+0C8h]
1400019FF: cmp  rax,0FFFFFFFFFFFFFDEEh
140001A05: ja   1400019B6
```

Listing 4.3: Annotation compiled in Visual Studio 2012.

4.4 Inference-Based Optimization

The requirement to specially compile the target application with annotations can be avoided by inferring the JIT code regions and instrumenting stores that frequently write to them. This section introduces INFERENCEDR, an extension of ORIGINALDR that includes the selective fragment removal from ANNOTATIONDR.

Parallel Memory Mapping Since there are no annotations to inform INFERENCEDR about JIT writes, it initially uses the detection scheme from ORIGINALDR. In addition, for each page A of memory from which dynamic code has been executed, INFERENCEDR associates a counter c_A with A and increments c_A on every write fault taken against A. When c_A exceeds a configurable threshold (default 12), INFERENCEDR creates a parallel mapping A' with writable permission. Since both the original page of memory and the new parallel mapping access the same underlying physical memory, a write to the parallel page is immediately visible for reading at the original page address. Similar parallel mapping techniques appear in recent works [124, 103, 45].

For the current and every subsequent write fault taken against A from writing basic block w, INFERENCEDR instruments each store instruction s in w with a prologue that (1) identifies each page s_p overlapping the address range s_r that s is about to write, and (2) queries a hashtable for any page $P \in s_p$ that has been mapped to a parallel page P'. For every such pair $\langle P, P' \rangle$, the prologue continues:

1. Lookup s_r in the fragment overlap hashtable (Sec. 4.2.3). If any fragments have been translated from s_r:
    (a) Exit to DynamoRIO and flush the stale fragments.
2. Replace s_r (in P) with s'_r (in P') so s can write freely.

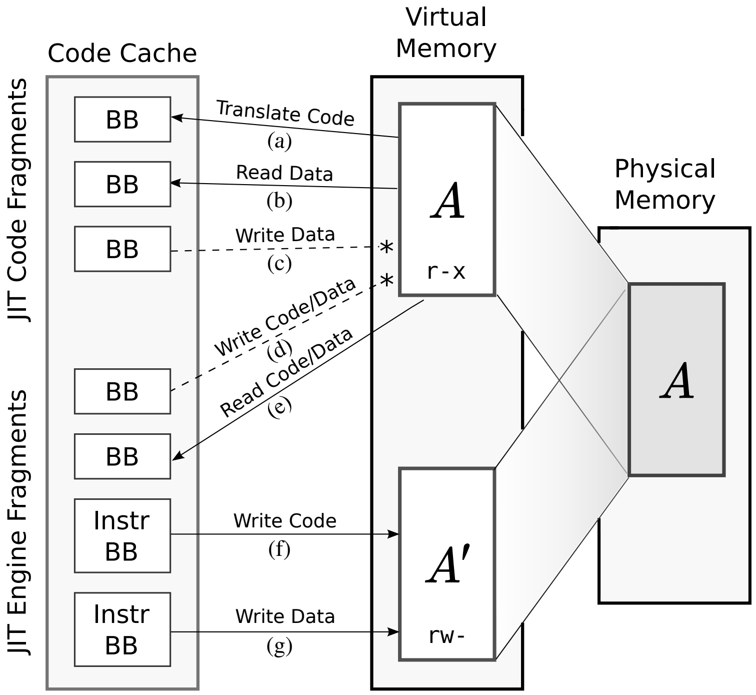


Figure 4.4: Parallel mapping in InferenceDR. Physical page A is mapped both to virtual page A and A', such that a write to A' is equivalent to a write to A.

Figure 4.4 illustrates a parallel mapping A and A', along with reads from A (arrows a, b and e), faulting writes to A (arrows c and d), and instrumented stores to A via A' (arrows f and g). On deallocation of A, InferenceDR removes both c_A and A'. These techniques combine to (1) eliminate the *concurrent writer problem*, and (2) avoid redundant invocation of the fragment flushing routine.

Eliminating the Concurrent Writer Problem Without annotations, detecting modifications requires not only that writable+executable memory pages be marked read-only, but also that the read-only status be maintained even if the executable permission is removed. Otherwise code fragments from the entire region would have to be flushed when the pages are set executable again, because any code changes would have gone unseen. This greatly increases the impact of the *concurrent writer problem*, making it especially significant for all JIT writes to be quickly instrumented with the redirection to the parallel page. Fortunately, large JIT engines tend to be well organized, and we find that a small number of stores are responsible for all the JIT writes.

Avoiding Redundant Fragment Flushing Even without the *concurrent writer problem*, extraneous invocations of the fragment flushing routine will generate significant overhead. The DynamoRIO clean-call facility allows any internal function to be called from the code cache by making a soft context switch, and this facility is used to invoke the same selective fragment flushing routine that was invoked by the annotation in Section 4.2. While the context switch is relatively efficient (approximately 100 CPU cycles), it becomes a significant expense when invoked for data writes in the JIT code area, which are frequent for highly optimized JITs like V8 and Ion. Redundant fragment flushing also occurs when the JIT overwrites code that was never executed—and hence never translated into the code cache. In both cases, the fragment overlap hashtable will contain no entries for the JIT write span, allowing the flushing clean-call to be skipped.

Constructing the Parallel Mapping On Linux it is not generally possible to create the parallel mapping from the target application's original memory allocation, because a shared memory file descriptor is required to attach the second mapping to the physical memory of the first mapping (a memory allocation is typically made via `libc` function `malloc()`, which does not use shared memory). Instead, INFERENCEDR maps a new shared memory page, copies the contents of the original page to it, and replaces the original page with a new parallel mapping of the copy. In the unusual case that the application has already setup a parallel mapping of its own, it would be possible for INFERENCEDR to query the OS for the location of the mapping and (if necessary) replace it with a specialized form of our parallel mapping that additionally maintains the application's original mapping.

JIT vs. Trampoline Both of the DGC strategies presented in Sections 4.2 and 4.4 are specifically optimized for large JIT engines that generate megabytes of code in a single execution. These optimizations work equally well for smaller JIT engines such as the Microsoft managed runtime, which generates small trampolines to bind application components to system libraries and services at runtime. The trampolines range in size from 4 bytes to roughly 250 basic blocks, so DynamoRIO will in some cases designate those trampoline regions as JIT code regions. While the frequency of code changes in these trampolines is much lower than for a JavaScript JIT engine, the *concurrent writer problem* has a significant enough impact to warrant parallel mapping the small number of pages occupied by the trampolines.

Implicit JIT Code Region Expansion When an instrumented store targets a page that has been marked read-only by INFERENCEDR for code change detection, the instrumentation is of no advantage—there is no parallel mapping for the page yet, so the concurrent writer problem still requires the entire region to be flushed. Since any instrumented store is already known to write JIT code, it is more likely that any executable page it writes to also contains JIT code. To minimize the overhead of region flushing, INFERENCEDR eagerly expands the JIT code region to include the written page. INFERENCEDR additionally avoids the page fault by placing read-only markers in the parallel mapping hashtable, allowing the instrumentation to determine that its store is about to fault, and instead make a clean-call to emulate the write and flush the region.

4.4.1 Parallel Mapping on Other Platforms

The basic technique for parallel mapping on Linux can also be applied for DBT running under Windows, though the procedure is slightly more complex. Allocating memory in Windows is a two stage process that requires first reserving the virtual address space and then committing physical storage to back the virtual address space. One complication is that portions of the reserved memory may be committed separately. Thus when a DBT engine discovers that a given reserve contains JIT code, the straightforward implementation of the DBT would simply copy the entire memory region and perform the parallel remapping. We expect that INFERENCEDR would port naturally to Mac OS X, which is based on Linux and supported by ORIGINALDR.

4.5 Evaluation

We evaluated the performance of each optimization stage from ORIGINALDR to INFERENCEDR relative to native performance on the Octane and Kraken [118] JavaScript benchmarks for two popular JIT engines, Chrome V8 and Mozilla Ion. We do not report results for SunSpider as it has largely been made obsolete by advances in JITs. We focus on JavaScript JITs because (1) other popular JITs such as Android's Dalvik and the Microsoft Managed Runtime are not available for our target platform, (2) JITs for higher-level scripting languages like Lua and PHP do not have well-established performance benchmarks and (3) Java is not especially relevant for DBT because the Java platform has its own ecosystem of tools that operate at the JVM level where interesting information is more readily visible. Table 4.2 shows both the performance overhead and improvement of each optimization for both Octane and Kraken.

All reported overheads represent the geometric mean of 3 runs, with P value of the two-tailed Student's t-test [70] no more than 9.076E-7. The test platform for all benchmarks is Ubuntu 13.04 on an Intel Xeon E3-1245 v3 running at 3.40GHz with 16GB memory and solid state drives.

Performance Improvement Figures 4.5 and 4.6 show the speedup of INFERENCEDR over ORIGINALDR and Pin (the performance of ANNOTATIONDR with selective fragment removal is nearly identical to INFERENCEDR). While our technique is applicable to DBT in general, other platforms such as QEMU and Valgrind are omitted from these results because they incur enough other overhead that DGC handling is not a bottleneck. Librando reports 3.5× overhead on Octane to randomize the V8 output for improved security. We focus our comparison on Octane here, but the relative performance of Pin is similar on Kraken.

Figures 4.5 and 4.6 additionally illustrate that INFERENCEDR performs much better on some benchmarks in the Octane suite than others. Runtime analysis reveals that the benchmarks for which INFERENCEDR performs best are getting the most mileage out of their compiled code, whereas

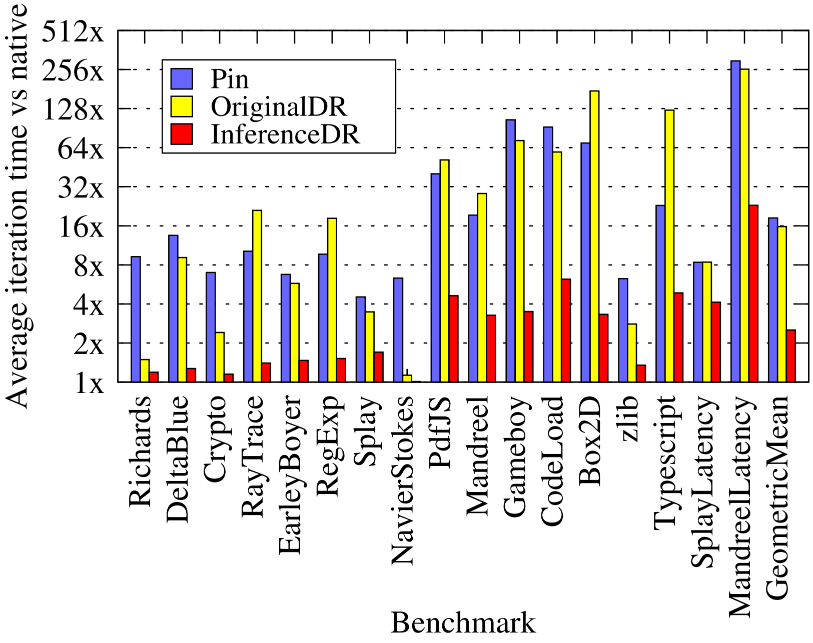


Figure 4.5: Optimization performance for Octane on V8.

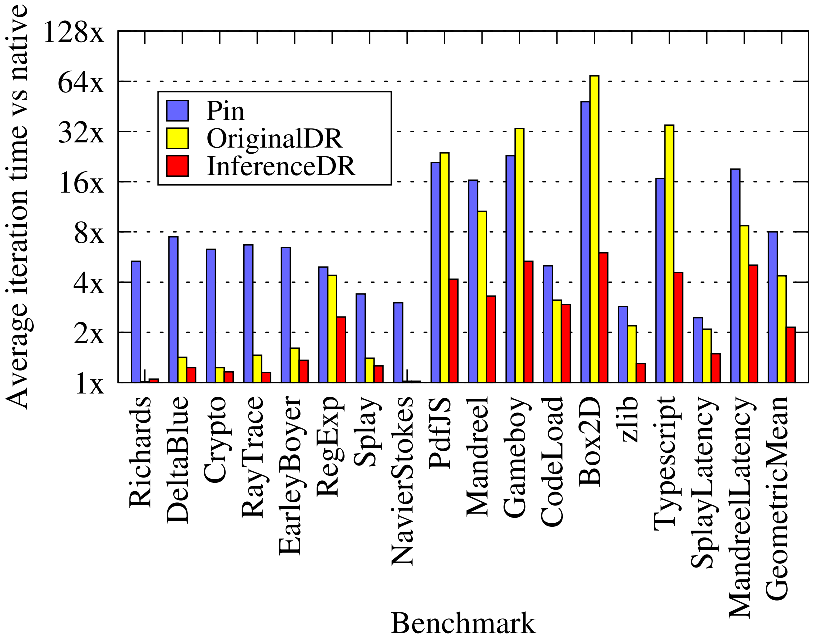


Figure 4.6: Optimization performance for Octane on Ion.

more cumbersome benchmarks like CodeLoad and Typescript compile more code and execute it fewer times. The worst case for INFERENCEDR is the two latency scores, which report special

	Octane Suite						**Kraken Suite**					
	Chrome V8			Mozilla Ion			Chrome V8			Mozilla Ion		
	Score	Overhead	Speedup	Score	Overhead	Speedup	Time	Overhead	Speedup	Time	Overhead	Speedup
OriginalDR	2271	15.80×	-	7185	4.36×	-	119s	9.08×	-	36s	3.25×	-
AnnotationDR + VM Area Isolation	9514	3.77×	4.19×	11914	2.63×	1.66×	35s	2.67×	3.4×	24s	2.14×	1.5×
AnnotationDR + Fragment Removal	14532	2.47×	6.40×	13797	2.27×	1.92×	24s	1.87×	4.96×	22s	1.95×	1.64×
InferenceDR	14257	2.52×	6.28×	14589	2.15×	2.03×	23s	1.78×	5.17×	23s	2.11×	1.57×
Native	35889	-	-	31340	-	-	13s	-	-	11s	-	-

Table 4.2: Performance improvement through several stages of DynamoRIO JIT optimization, as demonstrated in the Octane and Kraken JavaScript benchmark suites for both Chrome V8 and Mozilla Ion. Overhead is relative to native, and speedup is relative to OriginalDR.

measurements taken during the execution of Splay and Mandreel. SplayLatency isolates garbage collection time, while MandreelLatency individually times compile-intensive operations and squares the duration of each timed event before accumulation to penalize delays during these operations. This creates a worst-case scenario for InferenceDR, since its overhead is incurred during the operations that are penalized for latency.

Comparison of Approaches While InferenceDR and AnnotationDR offer similar performance, they minimize overhead in slightly different ways. Both approaches are able to isolate the specific code fragments that are modified by a JIT write, but only InferenceDR is able to avoid exiting the code cache in the common case that the JIT modifies or removes code that was never translated. Since a single store instruction only modifies a small number of bytes, the InferenceDR instrumentation can easily query the hashtable of overlapping code fragments from within the code cache. But the annotation may specify a JIT write spanning many pages of memory, making it significantly more complex (though not impossible) to perform the overlap check from within the code cache. This weakness in AnnotationDR is also its advantage, because it can remove a very large span of stale code fragments during a single cache exit. For example, Chrome V8 often generates several pages of code into a non-executable buffer and then copies it into the JIT code area. Since InferenceDR instruments low-level store instructions in the `memcpy` function, it must invoke the flush operation for each 16-byte write. Furthermore, shared functions like `memcpy` are used for many purposes other than copying JIT code, so while the overlap check is very efficient, the vast majority of overlap checks for the `memcpy` stores find no fragments to flush. Conversely, since the annotation of the JIT copy function is placed at a much higher level within the JIT code, ANNOTATIONDR is able to flush the entire set of pages in a single exit of the code cache.

While the annotations represent a contribution in themselves, ANNOTATIONDR would typically only be preferable in a scenario where INFERENCEDR is not feasible, since they offer similar performance. One example of such a scenario is that INFERENCEDR may not be fully compatible with 32-bit applications that can consume large amounts of memory, because during a memory-intensive execution there may not be sufficient address space to construct the parallel memory mappings. There is also a special case on the 32-bit Windows platform in which the application could initially allocate all of its memory in one very large reserve, requiring INFERENCEDR to double-map the entire reserve (Section 4.4.1)—which may not be possible in the 4GB address space of a 32-bit process. Should this become an issue, the user can choose to either (1) avoid such large workloads when running under the DBT tool, or (2) annotate the application and use ANNOTATIONDR.

Another consideration is security—the presence of the parallel mapping in virtual memory makes it possible for an adversary to write to the JIT code region undetected by the DBT. This could potentially create a vulnerability for a security-focused client of INFERENCEDR that needs to receive notifications of all DGC writes. Since protecting the parallel mappings inevitably creates additional overhead, ANNOTATIONDR may be the better alternative.

Annotation Developer Effort The amount of work required to annotate a target application varies. Annotating Ion required a complex runtime analysis to place 17 annotations. V8 was much easier because it is designed for portability to platforms that require an explicit i-cache flush—we only needed to annotate the global flush function in one place, and memory handling functions in 4 places, all of which were easy to find with no runtime analysis. For builds of V8 having Valgrind annotations enabled, ANNOTATIONDR can use the existing Valgrind annotation of the i-cache flush function, making the integration that much easier.

Space Overhead Both optimization approaches maintain hashtables that consume up to 2× the total size of all live DGC fragments, which is a moderate overhead in comparison to application data. INFERENCEDR additionally doubles the number of virtual memory mappings required for JIT code regions, but this does not increase physical memory usage (beyond OS accounting for the mappings).

	SPEC CPU 2006	SPEC Int	SPEC fp
ORIGINALDR	12.27%	17.73%	8.60%
INFERENCEDR	12.35%	17.88%	8.60%

Table 4.3: INFERENCEDR does not increase overhead for normal applications that do not dynamically generate code. ANNOTATIONDR is omitted because INFERENCEDR includes annotation detection (for the benefit of tools).

Negligible Side Effects We show that the optimization does not negatively impact the performance of DynamoRIO on normal applications by evaluating INFERENCEDR relative to ORIGINALDR on the SPEC CPU 2006 benchmark suite (Table 4.3). The SPEC CPU 2006 benchmarks include a broad range of applications written in C, C++, and Fortran, but does not include any dynamically generated code.

Chapter 5

ZenIDS

The difference between Paradise and a real IID for the standard PHP interpreter[1] lies simply in the four IID challenges (Section 1.2) along with the following three snares (Section 2.1) introduced by the PHP platform.

$S_{\textbf{cf}}$ In its zeal to provide the best in programming convenience, PHP has come to support a broad variety of indirect and sometimes esoteric control flow constructs. This would make an overly specific Trusted Profile prone to both false positives and false negatives.

- Destructors are invoked implicitly at some time after an allocated object goes out of scope. The invocation may depend on garbage collector state, making it effectively non-deterministic from the standpoint of user code.
- When a script attempts to read a non-existent property, PHP looks for method `__get()` on the object to call as a fallback, passing the name of the inaccessible property. Similarly for failed writes, the fallback is `__set()`. One effect of this feature is that many kinds of expressions can become call sites. For example, a trivial increment `foo->faux++;` can invoke both the `__get()` and `__set()` methods.

[1] `https://github.com/php/php-src`

- A user-defined object can override many of the PHP built-in functions. For example, if `foo->count()` is invoked on an object of class `Foo` that implements the `Countable` interface and defines a `count()` method, PHP will call it. Otherwise it calls the built-in `count()` function, passing the object as an argument.
- When a PHP script contains code outside of any function or method, the interpreter executes the code each time the file is loaded via `include` or `require`.
- PHP supports conventional try/catch semantics similar to Java and C++.

S_{cc} PHP supports dynamic evaluation of a string variable as PHP code. A popular idiom in PHP frameworks is the template engine, which dynamically generates PHP source files on disk based on fill-in-the-blank template strings. It is also possible for the source files of a PHP application to change at any time, since the source is interpreted on each execution, but this is not a conventional PHP idiom.

S_{late} One of the PHP ideals is to minimize the development of new features and maximize composition of existing features. To this end, the PHP development ecosystem thrives on massive frameworks that provide every imaginable component in an intuitive API. Many applications go one step further and expose these features to the site administrator by providing a configuration console that can activate any number of these dormant features with just a few clicks. Third-party plugins and manual code customization also contribute to late feature integration. For the purposes of ZENIDS, this has a similar effect to dynamically generated code.

Figure 5.1 depicts an overview of the components that have been added to Paradise so that ZENIDS can survive these challenges and snares. The threat model from Paradise is extended for the PHP application environment in Section 5.1. Then Section 5.2 presents an overview of the PHP interpreter and the ZENIDS extension, along with its format for the Trusted Profile in Section 5.2.1, which accounts for S_{cf} and S_{cc}. Next the attack surface and corresponding ZENIDS defense techniques are presented in Section 5.2.2. To account for S_{late}, Section 5.3 presents the evolution feature that automatically expands the Trusted Profile when a privileged user makes changes to the site. Section 5.5.4 presents performance optimizations to minimize the overhead of monitoring and the evolution feature. To conclude, experimental evaluation of the IID goals is presented in Section 5.5, including an assessment of the development and maintenance cost of ZENIDS.

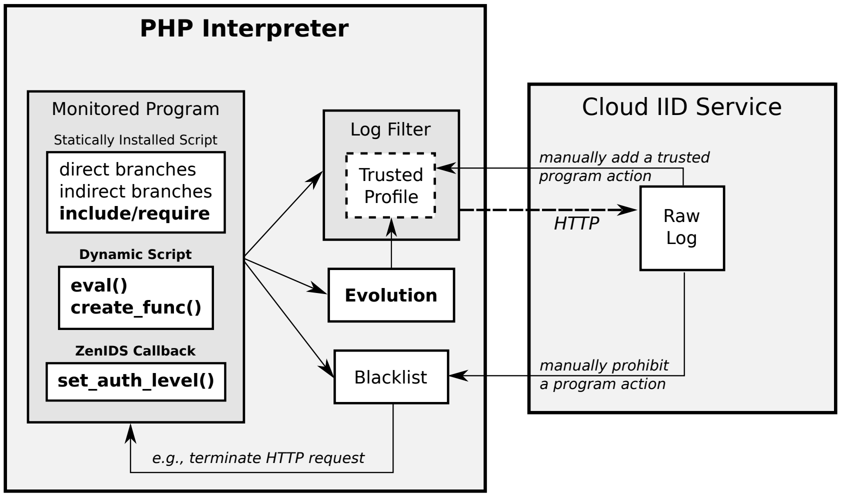


Figure 5.1: Component connectivity overview of application monitoring in ZENIDS. Bold font indicates components that were **not necessary in Paradise** (Chapter 2), but are necessary in PHP to account for the four IID challenges (Section 1.2) and three snares arising from PHP semantics and its development ecosystem (Section 2.1).

5.1 Threat Model

The Paradise threat model is extended to cover important aspects of the PHP environment. ZENIDS is designed to defend a PHP web application against a typical remote adversary who does not have login credentials, but may attempt to open a connection on the web server using any port and protocol available to the general public. The adversary can determine the exact version of the protected PHP application and the PHP interpreter, including ZENIDS, and has obtained the complete source code. A binary attack on the PHP interpreter cannot change the execution of the script other than to crash the process.

The adversary does not know when or where the ZENIDS profiling occurs, and is not able to access the Trusted Profile without first compromising the protected application. Web server authentication prevents untrusted uploads, except as permitted by the protected application itself. At the time of installation, the adversary had no opportunity to modify any files of the protected application; i.e., we assume the original installation is free of any backdoors that were specifically deployed by the adversary as part of the attack. There may, however, be backdoors in the application that are discoverable by the adversary at the time of its public source code release.

5.2 Monitoring

ZENIDS is implemented as a PHP extension supported by 8 callback hooks instrumented in the interpreter. Since S_{sem} does not affect the PHP interpreter, it is sufficient to focus detection on a call graph, augmented with intra-procedural control flow only for the purpose of distinguishing call sites. Section 5.2.1 outlines the reduction of PHP control flow to the call graph used in its Trusted Profile, along with the intra-procedural details required for automatic expansion of the Trusted Profile (Section 5.3). Whereas the high granularity of monitoring in BlackBox made it prone to false positives, monitoring a trivial call graph makes ZENIDS more susceptible to false negatives. To substantiate the effectiveness of this design against real exploits, Section 5.2.2 describes the ZENIDS architecture and presents several of today's most important RCE attack vectors to illustrate how it systematically detects malicious intrusions.

5.2.1 Trusted Profile

Although ZENIDS has the luxury of focusing on inter-procedural control flow, the effects of S_{cf} preclude modeling PHP execution with the simple call graph from Paradise. Inter-procedural control flow in PHP includes numerous edge types, along with unusual control flow such as multiple callees invoked in immediate succession from a single call site. Fortunately these edge types can be inferred from context, for example a destructor call is always made from a specific handler within the interpreter, where the callee is always a user-defined method named `__destruct()`. Since IID monitors without context sensitivity, any Trusted Profile will automatically flatten the multiplexed calls. Together, these observations make it possible for ZENIDS to transform the inter-procedural execution of PHP into a trivial call graph augmented with conventional exceptions.

Since PHP applications typically serve multiple users and distinguish permissions by an authentication mechanism, ZENIDS supports degrees of trust based on the privilege level of the logged-in user. This feature is optional and requires a small code modification in the monitored PHP application. The administrator instruments the application with callbacks indicating logins and logout by adding a call to a built-in function `set_user_level()` provided by the ZENIDS extension. For applications with complex privilege hierarchies, the argument to this function can be a lattice element, but in the ZENIDS prototype a simple integer suffices. ZENIDS stores the authentication level in the PHP session to make it available across requests and across server nodes. In the Trusted Profile, each edge is annotated with the lowest authentication level that is trusted to execute the corresponding control flow branch.

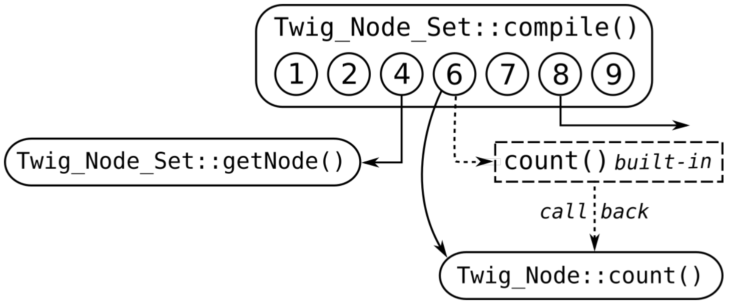
The Trusted Profile expansion feature is also optional, and in addition to the login instrumentation, it requires a higher granularity in the Trusted Profile. To associate previously untrusted code with actions taken by an authenticated user, ZENIDS employs a taint tracking algorithm that examines trust information at each intra-procedural edge. Since the PHP interpreter executes a script by first compiling it into opcode sequences (e.g., one per function, method, or script body), ZENIDS can associate one CFG node to each opcode in a compiled sequence. Figure 5.2 illustrates this, starting from (a) the source code, which is compiled into (b) an opcode sequence, then recorded by ZENIDS during execution to (c) the trusted profile.

```
46 public function compile(Twig_Compiler $compiler)
47 {
48   if (count($this->getNode('names')) > 1) {
49     $compiler->write('list(');
```

(a) PHP code sample from the Symfony template compiler Twig.

#	Line	Result	Opcode	Operand 1	Operand 2
0	46	-	ZEND_RECV	`$compiler`	-
1	48	-	ZEND_INIT_FCALL	`count`	-
2	48	-	ZEND_INIT_METHOD_CALL	`getNode`	-
3	48	-	ZEND_SEND_VAL_EX	`"names"`	-
4	48	`tmp #1`	ZEND_DO_FCALL	-	-
5	48	-	ZEND_SEND_VAR	`tmp #1`	-
6	48	`tmp #2`	ZEND_DO_FCALL	-	-
7	48	`tmp #3`	ZEND_IS_SMALLER	`1`	`tmp #2`
8	48	-	ZEND_JMPZ	`tmp #3`	`+25`
9	49	-	ZEND_INIT_METHOD_CALL	`$compiler`	`write`

(b) The sample (a) compiled by PHP into an opcode sequence.



(c) The trusted profile CFG of the opcode sequence (b). At opcode #6, the callee `count()` is a built-in function that either counts built-in collections such as arrays, or calls back to the object's `count()` method if it has one. ZENIDS models the callback as an edge connecting opcode #6 directly to `Twig_Node::count()` to avoid having hundreds of edges from a nexus `count()` node.

Figure 5.2: Execution of a PHP code snippet under ZENIDS profiling.

Figure 5.3 depicts the node and edge types represented in the ZENIDS Trusted Profile. Since PHP enforces procedure boundaries, the Trusted Profile accordingly defines a `Procedure` type that contains the corresponding compiled opcode sequence. It is not generally possible to group opcodes into basic blocks in PHP, since the majority of opcodes may implicitly invoke a procedure, and static analysis of the callees is intractable (see Section 5.2.3). An exception edge is implied by an Inter-Procedural Edge having a `to_node` that is not the procedure entry point. Several artifacts of

S_{cf} can weaken the Trusted Profile, leading to false positives or negatives during monitoring. To avoid this, ZENIDS implements special cases in the Trusted Profile:

- When the interpreter initiates a callback to a user-defined function, such as a destructor, ZENIDS represents the call site with a symbolic *system* node. Even though the physical call stack might suggest that the destructor is invoked at a certain opcode, for example where the object goes out of scope, this is not consistent across executions.
- As depicted in Figure 5.2, when user code calls a built-in function, it may in turn invoke a callback. For commonly used built-ins like `count()`, this would create a nexus that effectively aliases all callback callees of `count()`. ZENIDS preserves this distinction by representing the callback as an edge directly from the invocation of the built-in to the callback callee.

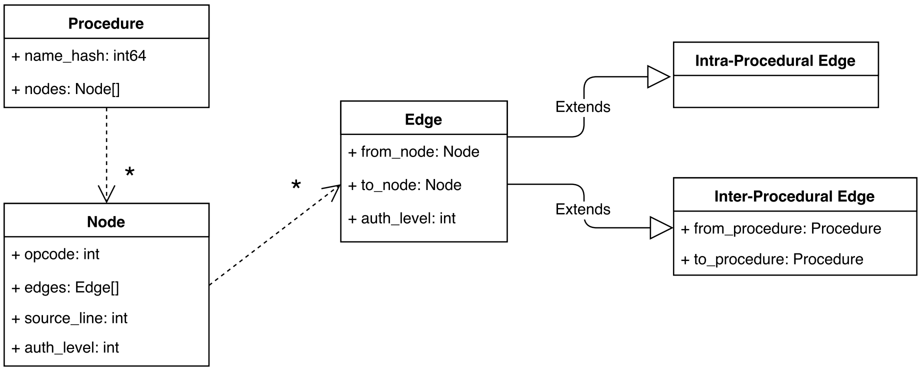


Figure 5.3: ZENIDS Trusted Profile (simplified).

G_{kx} Avoiding Implicit Aliasing

The transformations from PHP control flow to the ZENIDS Trusted Profile are essential for maintaining accurate detection of attacks. For example, in an exploit of the WordPress Download Manager plugin [177], the adversary creates a new privileged user by manipulating the PHP built-in `call_user_func()` to target the WordPress core function `wp_create_user()`. If ZENIDS allowed a built-in to become a nexus, it would not know which call site normally (safely) reaches which callee, making this exploit potentially undetectable.

5.2.2 Detection

This section presents the ZENIDS architecture and illustrates how its low granularity of monitoring remains sufficient for detecting RCE attacks against PHP applications. Figure 5.4 shows the main components of the web server, as configured for our experiments, with the ZENIDS hooks labeled **H1**-**H5** (the other 3 hooks serve trivial initialization purposes). ZENIDS only relies on the first two hooks to detect attacks: **H1** correlates each compiled opcode sequence with its trusted profile (if the sequence is trusted at all), and **H2** validates each invocation of an opcode sequence (i.e., a procedure call) by matching it to an edge in the trusted profile. Listings **H1** and **H2** present the essential functionality of these hooks in pseudo-code.

Since the PHP interpreter does not assign a consistent identifier to dynamically evaluated script fragments (i.e., `eval()` and similar), in this case the sequence of opcodes itself is used as key in **H1** and **H2**. While this is much more performance intensive than the simple hash comparison for opcode sequences that are compiled from PHP files, it occurs rarely enough that observed overhead is minimal.

A PHP deployment is not limited to the typical configuration that we use for our experiments—it may incorporate numerous extensions, interact with external data sources and services, and be distributed across multiple servers having various architectures and operating systems. But these factors do not interfere with the ZENIDS detection mechanism. At its core, the PHP interpreter is simply a recursive iterator over opcode sequences, and to our knowledge there is no configuration that substitutes or modifies the central Opcode Execution Loop. For this reason, we expect the fundamental approach of ZENIDS to be compatible with PHP deployments of all varieties.

PHP Interpreter Hook `H1` Compile PHP code into *op_seq*

```
$key \leftarrow$ CANONICAL-NAME($op\_seq$)
if $key \in trusted\_seqs.keys$ then
    $trusted\_seq \leftarrow trusted\_seqs.get(key)$
    if IS-IDENTICAL($op\_seq, trusted\_seq$) then
        $op\_seq.trusted\_seq \leftarrow trusted\_seq$
    else
        REPORT($untrusted\_op\_sequence$)
    end if
end if
```

PHP Interpreter Hook `H2` Enter *target_seq* from *op_seq*[*i*]

```
if $op\_seq[i].target\_seq.trusted\_seq = $ NIL then
    REPORT($untrusted\_app\_entry\_point$)
else
    $key \leftarrow$ CANONICAL-NAME($op\_seq[i].target\_seq$)
    if $key \notin op\_seq[i].trusted\_targets$ then
        REPORT($untrusted\_call$)
    end if
end if
```

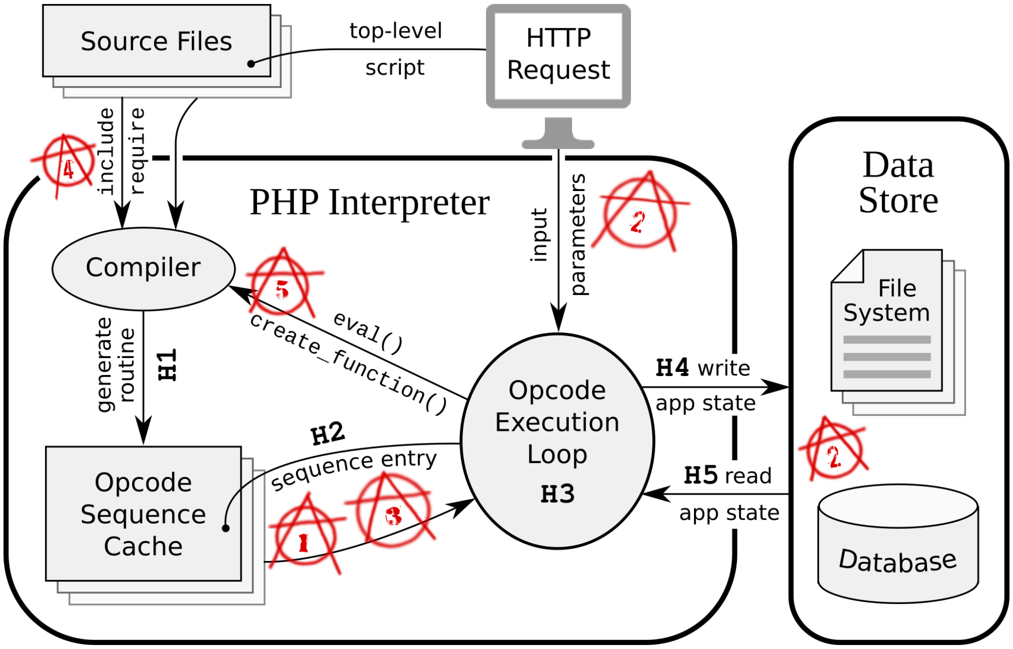


Figure 5.4: Components of a typical PHP deployment along with ZENIDS hooks **`H1`-`H5`** and fundamental RCE attack vectors **`A1`-`A5`**.

Canonical Names

For **`H1`** and **`H2`** to be reliable, CANONICAL-NAME() must be consistent across executions, since ZENIDS uses it to find trusted profile entries. While it might be trivial in many languages, PHP raises several complications. To begin with, the `include` and `require` keywords are compiled as statements and executed in the opcode sequence (not pre-processed like in C/C++), making it possible for conditional branches to govern the set of imported source files. Function declarations similarly allow for conditional compilation at runtime. Combining these factors with namespaces, it is possible—and very typical—for a PHP application to define the same function identifier multiple times, often among many different files (e.g., to implement a plugin interface). ZENIDS avoids ambiguity in the canonical name using the following schemes:

- $\langle$filename$\rangle$.$\langle$function-name$\rangle$.$\langle$line-number$\rangle$
- $\langle$filename$\rangle$.$\langle$classname$\rangle$.$\langle$method-name$\rangle$.$\langle$line-number$\rangle$

Attacks

Figure 5.4 also labels five important attack vectors taken by today's corpus of RCE exploits (**`A1`**-**`A5`**). The pivotal role of the Opcode Execution Loop in the PHP interpreter makes it possible for ZENIDS to detect all five vectors in hook **`H2`**:

A1 Call By Name: When a function callee is specified as a string constant or variable, PHP resolves the callee using the set of functions defined dynamically during execution. An exploit of the popular WordPress Download Manager plugin creates a new user with administrator privileges by manipulating the target of just one PHP call-by-name [177]. ZENIDS reports an anomaly at **H2** on any untrusted call edge, even if the call site and the callee are in the trusted profile.

A2 Object Injection: The format of serialized objects in PHP specifies both the type and content of each field, making it possible for the adversary to compose arbitrary serialized instances. Dozens of object injection attacks have been reported, such as CVE-2015-8562 against Joomla in which the adversary executes arbitrary code by fabricating a serialized session. In this scenario ZENIDS will detect untrusted edges in the payload at **H2**.

A3 Magic Methods: PHP implicitly invokes specially named "magic methods" in certain situations, for example a call to an undefined method `$a->foo()` is forwarded to `$a->__call()` with the name of the missing callee and the arguments (as a fallback). Esser and Dahse combine *object injection* with *magic methods* to create *Property-Oriented Programming* attacks [52, 41] that can execute arbitrary code. While the approach is more complex than **A2**, the ZENIDS defense at **H2** remains the same.

A4 Dynamically Included Code: The PHP `include` and `require` statements can take a string variable argument, allowing an intruder to import any file. Since ZENIDS models these as calls, **H2** will detect untrusted targets.

A5 Dynamically Interpreted Code: PHP can execute a plain text string as code, making the input string vulnerable to attack. ZENIDS models these dynamic opcode sequences as dynamic imports and monitors them at **H2**.

$G_{\mathbf{rx}}$ Case Study: Object Injection Attacks

Although recent research has developed many sophisticated algorithms for finding and exploiting vector **A2** in PHP applications, the attacks themselves are no more difficult for ZENIDS than the most rudimentary unsanitized input. This is not true for defense techniques that attempt to constrain these dynamic constructs on the basis of static analysis. The vulnerability behind any **A2** attack is a dataflow from unsanitized input to a call-by-name target or `eval()` string. To determine safe values at these sites, static analysis must either (a) assume that any possible string may reach the vulnerability, or (b) model some external construct such as the PHP session or database tables. But the IID approach does not have any such limitation. Whether a malicious object is injected directly through an HTTP parameter, or follows a lengthy maze of database accesses, ZENIDS still detects the anomaly in **H2** at the compromised call-by-name or `eval()` site.

CVE-2015-7808

Figure 5.5 shows a simplified version of the vulnerable class `vB_dB_Result` (left), which is serialized by PHP into an encoded string (right top). Since PHP assigns all fields during serialization, the adversary can create a malicious `vB_dB_Result` (right bottom) that invokes `shell_exec("whoami");` in the destructor. Since the `cleanup` field is assigned via deserialization, static analysis cannot determine the legal set of function names. But the Trusted Profile will contain an edge to the trusted cleanup function(s), which for this application will certainly never include `shell_exec()`.

5.2.3 Dynamic vs. Static Analysis

If it were possible for a static analysis to determine all normal execution paths in a PHP application, S_{late} would not be such a significant snare, because the most common late binding scenario is the

```
class vB_dB_Result {
  var $cleanup = 'mysql_free_result';
  var $recordset;

  function __destruct() {
    call_user_func($this->cleanup,
                   $this->recordset);
  }
}
```

```
O:12:"vB_dB_Result":2:{
  s:7:"cleanup"; s:17:"mysql_free_result";
  s:9:"recordset"; N;
}
```

```
O:12:"vB_dB_Result":2:{
  s:7:"cleanup";s:10:"shell_exec";
  s:9:"recordset";s:6:"whoami";
}
```

Figure 5.5: Object forgery attack against vBulletin. The vulnerable `vB_dB_Result` class on the left is serialized by PHP to the encoded string on the right (top). An adversary manipulated the serialized string into the form on the right (bottom), which resulted in a call to `shell_exec()` instead of the expected `mysql_free_result` function (or similar) in the destructor.

activation of framework features, which could all be analyzed on disk. Having access to the source code might seem to make PHP an ideal candidate for static analysis. But the presence of S_{cf} makes the call graph difficult to determine statically, even from the source code. Listing 5.1 shows a simple case of an expression that may or may not be a call site, depending on the session state at runtime. For static analysis to be accurate, it must perfectly model all possible session states, which is known to be intractable.

Listing 5.1: Static analysis cannot easily determine whether the field access `$a->size` on line 10 will call a function, because the type of `$a` is dynamically specified in the session, which persists values across HTTP requests.

```
 1  class Foo {
 2    public function __get($field_name) {
 3      echo("Unknown field " . $field_name);
 4    }
 5  }
 6  $a_type = $_SESSION["type-of-a"];
 7
 8  $a = new $a_type();
 9
10  echo("a.size = " . $a->size . "\n");
```

5.3 Supporting Website Evolution

A popular design idiom for PHP applications is to provide a large set of dormant features that a privileged user can easily enable and configure through an admin console. Another popular idiom generates new PHP source files at runtime on the basis of fill-in-the-blank template. While these idioms are advantageous for both developers and users, the corresponding changes in control flow create a snare for IID, particularly in the legacy profiling and end-user profiling scenarios. These idioms effectively introduce untrusted code into the application that causes a high rate of spurious anomalies. A variation of this snare is the proliferation of plugins and other third-party additions throughout the PHP ecosystem. This also affects the vendor profiling scenario, because it may not be possible for a development organization to maintain Trusted Profiles for all of the components that may be integrated with their product. A further extreme is the custom integration of components where there is no plugin interface in the original core product, which occurs frequently among ambitious site maintainers.

To account for the S_{late} binding of new features, ZENIDS responds to trusted configuration changes by expanding the trusted profile to enable relevant new code paths. When ZENIDS detects that a privileged user has changed application state, for example in the database, it initiates a *data expansion event* to add corresponding new control flow to the trusted profile. Similarly, if the application generates or modifies PHP source files in a way that conforms to a trusted code generator, ZENIDS initiates a *code expansion event*. The duration of an expansion is limited to a configurable number of requests (2,000 in our experiments) to minimize an adversary's opportunity to manipulate the expansion into trusting unsafe control flow. Hook **H3** initiates expansions on the basis of the sets $tainted\_values$ and $safe\_new\_sources$, which are maintained by system call monitors **H4** and **H5**.

Listing **H3** shows how ZENIDS adds edges to the trusted profile when the expansion conditions are met. The first condition initiates a data expansion (lines 4-5) at an untrusted branch decision

PHP Interpreter Hook `H3` Execute i^{th} opcode of op_seq

```
if $mode = monitoring$ then
    if Is-Branch($op\_seq[i]$) &
            $branch \notin op\_seq[i].trusted\_targets$ then
        if $branch.predicate \in tainted\_values$ then
            $mode \leftarrow expanding$
            $trust\_to \leftarrow$ Control-Flow-Join($branch$)
        else if Is-Call($branch$) then
            if $call.target\_seq \in safe\_new\_sources$ then
                $mode \leftarrow expanding$
                $trust\_to \leftarrow i + 1$
            else
                Report($untrusted\_call$)
            end if
        end if
    end if
else                                                    ▷ $mode = expanding$
    if Is-Branch($prev\_op$) then
        $prev\_op.trusted\_targets \cup \{op\_seq[i]\}$
    end if
    if $i = trust\_to$ then
        $mode \leftarrow monitoring$
    else if Is-Assignment($op\_seq[i]$) then
        $tainted\_values \cup \{op\_seq[i].predicate\}$
    end if
end if
Propagate-Taint($op\_seq, i, i + 1$)
```

(line 2) when at least one value in the branch predicate carries taint from an administrator's recent data change (line 3). The second condition initiates a code expansion (lines 8-9) when an untrusted call (lines 2 and 6) is made to a safe new source file (line 7). If neither expansion condition is met, and the branch is any kind of call, then an $untrusted\_call$ anomaly is reported (line 11). During an expansion, new branches are added to the trusted profile (line 17) and taint is propagated across all assignments (line 22) and uses of tainted operands (line 25). The expansion ends (line 20) where the initiating branch joins trusted control flow (line 19).

5.3.1 Code Expansion Events

Since PHP applications often have repetitive source code, many PHP libraries provide an API to generate new source files at runtime on the basis of application-defined templates.

PHP Interpreter Hook H4 Store application state

```
if IS-TRUSTED-CODE-GENERATOR() then
    $safe\_new\_sources \cup \{new\_source\}$
else if IS-ADMIN($user$) then
    $state\_taint \cup \{stored\_state \times user.admin\_level\}$
end if
```

PHP Interpreter Hook H5 Load application state

```
if !IS-ADMIN($user$) & $loaded\_state \in state\_taint$ then
    $tainted\_values \cup \{loaded\_state\}$
end if
```

For example, GitList uses the Symfony component Twig to define the HTML page layout for each type of repository page: file list, commit list, blame, etc. At runtime, Twig generates corresponding PHP files on demand. Incorporating this kind of dynamic code into the trusted profile is easy for ZENIDS if the code generator runs during the training period. But a demand-based code generator may run at any time—for example, in our GitList experiment (Section 5.5.2), crawlers found unvisited views several weeks after the 2-hour training period.

To continue trusting these known-safe code generators, each time the application persists a state change to the database or the filesystem, IS-TRUSTED-CODE-GENERATOR() in **H4** determines whether the application has just written a safe new source file. This function examines the following criteria:

1. **Call Stack:** At the API call to update the database or write to a file, does the call stack match any of the code generator call stacks recorded during the training period?
2. **User Input Taint:** If during training the application never generated source code using values influenced by user input, then ZENIDS checks whether the data for this state change carries taint from user input. This criterion tracks both directly and indirectly tainted values (and may be disabled to avoid continuous taint tracking overhead).
3. **Generator Visibility:** Hook **H4** additionally preserves a snapshot of the persisted data—if during training the application only generated source code via the PHP persistence API, then a new source file will only be trusted if it matches the last snapshot taken at **H4**.

5.3.2 Taint Tracking

The expansion events rely on propagation of taint from user input and authorized state changes. User input is tainted at HTTP input parameters and session values, while data loaded by the application is tainted in **H5** on the basis of $state\_taint$. Function PROPAGATE-TAINT() in **H3** (line 25) transfers both colors of taint across assignments, compositional operations such as arithmetic and comparisons, arguments to function calls, and return values from functions. Hook **H3** also implicitly taints all assignments that occur during an expansion (line 22), since those assignments have as much of a causal relationship to the taint source as the branch itself. But ZENIDS does not implicitly taint assignments within trusted code, even if it occurs under a tainted branch, because those assignments must have already been made at some time prior to the expansion event—before taint was present on the predicate—indicating that the influence represented by the taint is not pivotal to that already-trusted branch decision.

Taint Implementation

To avoid complication as taint flows through composite structures, for example the fields of an object, ZENIDS implements taint as a hash table entry using the physical address of the tainted value as the hash key. While it would be simpler to directly flag the internal operand struct, there are no free bits for primitive type operands, and expanding the 8-byte struct would cause tremendous complication throughout the interpreter. Directly tainting operands also simplifies integration into the PHP interpreter, where the majority of explicit taint transfers are implemented as a simple stack push within the interpreter's `ZVAL_COPY()` macro.

5.4 Performance Optimization

The overhead of branch evaluation in **`H2`** and taint tracking in **`H3`** (line 25) could make a naïve implementation of ZENIDS unusable in a high-traffic deployment. Even checking for the presence of taint in **`H3`** (line 3) can increase overhead by an order of magnitude. But since these expensive operations are only necessary in certain scenarios, it is safe for ZENIDS to elide them when conditions indicate that the operations will always nop. For maximum efficiency, ZENIDS implements three flavors of Opcode Execution Loop (Figure 5.4), each taking only the actions necessary for its designated context:

1. **MonitorAll**: Propagates taint, evaluates all branch targets.
    - Reserved for profile expansion events (2,000 requests).
2. **MonitorCalls**: only evaluates call targets.
    - This is the default for monitoring untrusted requests.
3. **MonitorOff**: ignores everything.
    - Reserved for trusted users (negligible overhead).

Since PHP invokes the Opcode Execution Loop via function pointer, switching is simply a matter of replacing the pointer. The two monitoring modes are further optimized as follows:

MonitorCalls The first 4 lines of **`H2`** are elided by lazily grafting the set of safe call targets onto each PHP opcode. ZENIDS cannot extend the 32-byte opcode struct because interpreter performance relies on byte alignment, so instead it borrows the upper bits of a pointer field (which are unused in 64-bit Linux where the user-mode address space is much smaller than 2^{64}). Specifically, a pointer into the trusted profile is copied into a cache-friendly array, whose index is packed into the spare bits of the opcode. Line 5 of **`H2`** is further optimized for call sites having only one trusted target: instead of a trusted profile pointer, the array entry is a direct encoding of the singleton target, allowing for bitwise evaluation. To avoid expensive string comparisons, target opcode sequences are identified by a hashcode of the canonical name.

MonitorAll Since ZENIDS maintains taint in a hashtable, the accesses required for taint propagation could become expensive under rapid iteration of hook **`H3`** (Figure 5.4). In addition, some PHP opcodes might require a costly callback from the interpreter because they affect complex data structures or are implemented using inline assembly. Both sources of overhead are alleviated by lazy taint propagation using a queue that is locally maintained by simple pointer arithmetic.

When the interpreter copies internal data structures (e.g., to expand a hashtable), taint must be tediously transferred to the new instance. ZENIDS flags each data structure that contains taint and elides this expensive transfer for untainted structures.

5.4.1 Synchronizing Evolution

For applications that store state in a database, the persisted $state\_taint$ (Listing **H4**) is updated by database triggers, since it is non-trivial to determine from an SQL string what database values may change (we assume instrumenting the database engine is undesirable). But the query for $state\_taint$ can be expensive relative to the total turnaround time of a very simple HTTP request, even when our query uses a trivial prepared statement. Instead, ZENIDS stores the request id of the last privileged state change in a flat file and updates it from the database every 20 requests. While this delay makes it possible to raise up to 20 false positives immediately after a state change, it is unlikely because a significant state change often involves multiple HTTP requests. For example, although the WordPress permalinks feature can be enabled with just one radio button selection, it takes at least 5 asynchronous HTTP requests to process that change, and only the last one enables the feature.

5.5 Experimental Evaluation

We conduct several controlled and real-world experiments to demonstrate the ability of ZENIDS to meet four of the quantitative IID goals. Section 5.5.1 pursues both G_{kx} and G_{dnd} with a controlled experiment to detect recently published exploits against WordPress components. Section 5.5.2 similarly pursues G_{wx} and G_{dnd} in the context of real PHP applications facing live Internet traffic, and Section 5.5.3 focuses on the evaluation phases of these experiments. Section 5.5.4 pursues G_{perf} with a benchmark based on live traffic, followed by qualitative discussions of G_{dep} and G_{dev}.

Since we were not able to obtain a working version of this exploit together with the corresponding vulnerable application, a case study on Property Oriented Programming was presented in Section 5.2.2. The qualitative evaluation of G_{dbg}, G_{mod} and G_{blk} are elided because the design and implementation of these features is effectively identical to BlackBox (see Section 3.6).

5.5.1 Monitoring a Vulnerable Application

We demonstrate that ZENIDS detects publicly reported exploits by configuring a WordPress site with vulnerabilities targeted by the 10 most recent WordPress exploits published on `exploit-db.com` (as of June 18, 2016), excluding data-only attacks and unavailable modules. This medley, shown in Table 5.1, shows that ZENIDS can detect a broad variety of exploit strategies. For authenticity we configured and exercised at least 30% of each module's high-level features. To train the trusted profile we invited common crawlers and manually browsed the site for just a few minutes using a range of mobile and desktop browsers. For modules having forms, we entered a variety of valid and invalid data and uploaded valid and invalid files. Then we enabled the ZENIDS monitor and continued using the site to evaluate our goals:

G_{kx} We invoked each exploit and observed that (a) ZENIDS reported the expected anomaly, and (b) the POC succeeded, indicating the intrusion report was true.

G_{dnd} False positives only occurred when entering invalid form data—all other requests were trusted by ZENIDS.

5.5.2 Monitoring Live Applications

The main goal of ZENIDS is to accurately detect exploits occurring in the wild while minimizing false positives. This experiment shows that ZENIDS can quickly learn the normal behaviors

EDB-ID	WordPress Module	Exploit Detected
39577	AB Test Plugin	✓
39552	Beauty & Clean Theme	✓
37754	Candidate Application Form Plugin	✓
39752	Ghost Plugin	✓
39969	Gravity Forms Plugin	✓
38861	Gwolle Guestbook Plugin	✓
39621	IMDb Profile Widget	✓
39883	Simple Backup Plugin	✓
37753	Simple Image Manipulator Plugin	✓
39891	WP Mobile Detector Plugin	✓

Table 5.1: ZENIDS reports an intrusion during attempts to exploit vulnerable plugins and themes in a WordPress site. These were the most recent 10 WordPress exploits from `exploit-db.com` as of June 18, 2016, excluding data-only attacks and unavailable modules.

of popular applications by simply profiling live Internet traffic, allowing ZENIDS to accurately distinguish exploits from normal (safe) user requests. But this approach may incorporate the application's handling of certain malicious behaviors into the Trusted Profile, for example invalid logins, such that ZENIDS does not report anomalies for these normal but unsafe inputs.

This experiment focuses on a year of live traffic to three PHP applications hosted by our research lab:

1. **WordPress:** Our lab website, which has 10 pages and uses the "Attitude" theme (with no posts or comments).
2. **GitList:** The public repository viewer for software developed in our lab, based on the Symfony framework.
3. **DokuWiki:** A computer science class website containing wiki-text markup, document previews and file downloads.

The applications were hosted on a standard PHP interpreter, and traffic was recorded for a period of 360 days so that the experiment could be conducted offline and accurately repeated. This approach allows selection of various profiling periods to study how that factor affects the accuracy of intrusion detection. Profiling was augmented by crawling each site with the utility `wget`, which is unaware of JavaScript and only follows URLs appearing directly in the HTTP response text.

Table 5.2 presents the results for the optimal profiling periods, which were selected in retrospect based on the effectiveness of the corresponding Trusted Profile. ZenIDS reported 38,076 anomalies on a diverse array of attacks using a broad range of exploit techniques,. This includes several attempts at the notorious XML RPC brute-force password exploit (where each RPC batch is reported as just one anomaly). The malicious nature of the requests was verified manually on the basis of common hacks (e.g., SQL appearing in the HTTP parameters) and published or well-known exploits. A large percentage of these attacks targeted applications that were not installed on the webserver, yet still caused control flow anomalies. The ratio of unique false positives to requests is well below .01% for all applications, where *unique* refers to distinction in control flow anomalies (i.e., two HTTP requests having the same set of anomalies are considered one unique anomaly report).

			Intrusion Reports				
	Requests		Total	False Positives		False Negatives	
	Total	Training		Unique	Rate	#	Rate
WordPress	248,813	595	36,693	3	.00001%	16679	.07%
GitList	1,629,407	298	1,744	1	<.000001%	0	0
DokuWiki	24,574	3,272	253	3	.0001%	0	0

Table 5.2: Intrusion attempts reported by ZenIDS while monitoring our lab web server for 360 days. False negatives represent safely handled attacks such as invalid logins, or attacks on applications that we do not host.

G_{wx} False Negatives and Live Profiling

The vast majority of false negatives represent (a) invalid logins, (b) unauthorized access to admin pages, and (c) attempts to access privileged functionality that did not exist on the server. In the sense that no successful exploit occurred during the experiment, there was no possibility of an absolute false negative. But we choose to report a false negative for any HTTP request that (a) exhibited malicious intent, (b) exercised control flow that is not essential for benevolent users, and (c) did not cause ZENIDS to report an anomaly. These could perhaps be called *gray negatives* because the application safely handled the HTTP request, responding either with a 404 page or redirection to the login page. It is likely that most of these HTTP requests were generated by malicious bots, and that they occur very frequently on any public server, even if it does not host any web applications—this is evident from the fact that all the gray negatives occur in WordPress, which receives all HTTP requests for URL forms that are not recognized by the Apache webserver. None of these requests would have succeeded in their malicious intent on any version of our applications, outside of guessing a WordPress password.

To understand the effect that the profiling period has on the high rate of gray negatives, Table 5.3 shows similar results for a range of profiling periods. An increase of 7× for WordPress to 4,128 training requests results in just a 6% increase in gray negatives, which totals less than .01% of the year's WordPress traffic. The additional 966 gray negatives have the same characteristics—they are evidently generated by bots and have no legitimate means of compromising any version of the installed applications. It would certainly be better to have supernatural test bots that generate a perfect Trusted Profile. But the research question behind these statistics is simply, how much accuracy is lost by profiling live Internet traffic? In the case of this experiment, ZENIDS loses visibility of the random malicious noise that has a low probability of success but is generated in such high volumes that it is likely to appear in any sample of HTTP requests. While 16,679 false negatives may seem significant, 16,679 gray negatives is effectively quite close to Paradise.

These experiments not only show that ZENIDS detects a broad range of attacks, but also that it integrates effectively into a diverse set of applications representing an important cross-section of today's PHP landscape. These frameworks serve a large percentage of the world's HTTP traffic and support millions of websites ranging in significance from personal homepages to household names in the Alexa top 100.

			Intrusion Reports			
	Training Requests	Total	False Positives (unique)		False Negatives	
			#	Rate	#	Rate
WordPress	595	36,693	507 (3)	.002% (.00001%)	16,679	.07%
	1,188	35,826	507 (3)	.002% (.00001%)	17,546	.07%
	2,070	35,767	507 (3)	.002% (.00001%)	17,605	.07%
	4,128	35,727	507 (3)	.002% (.00001%)	17,645	.07%
GitList	298	1,744	6 (1)	$<$ .000001%	0	0
	862	1,744	6 (1)	$<$ .000001%	0	0
	12,010	1,744	6 (1)	$<$ .000001%	0	0
	72,068	1,744	6 (1)	$<$ .000001%	0	0
DokuWiki	1,462	364	224 (55)	.009% (.002%)	0	0
	3,272	253	101 (3)	.004% (.0001%)	0	0
	4,133	253	101 (3)	.004% (.0001%)	0	0
	5,559	217	65 (2)	.003% (.0001%)	0	0

Table 5.3: The duration of the training period has minimal impact on the accuracy of ZenIDS detection. False positives in DokuWiki could potentially be avoided by improvements to ZenIDS. False negatives represent attacks that failed to have any effect on control flow.

G_{dnd} False Positives and Live Profiling

ZenIDS only raised 507 false positives (3 unique) against WordPress in the entire year. A broken link triggered 502 of them (2 unique) at the "guess permalink" function—one might argue that these are useful anomalies to report, since they do reflect an error in the site.

Improvements to ZenIDS could potentially eliminate all the false positives in DokuWiki. Half were caused by the addition of a tarball download to the wiki, which does not trigger an expansion event because new control flow occurs before the tainted tarball is loaded from the filesystem. ZenIDS could enable this expansion by tainting operands at untrusted branches and, after the request completes, checking backwards for privileged user influence. The remaining false positives were caused by crawlers reaching new pages, which could be avoided by blocking suspicious requests from crawlers.

We experienced just one false positive in over 1.5 million GitList requests despite training the trusted profile in just 2 hours—a total of 298 requests—highlighting the simplicity of our dynamic approach vs. a traditional static analysis. Our trusted profile for GitList covers 62 closures and several dynamically generated PHP files, along with 33 distinct magic methods reached from 54 call sites (excluding constructors and destructors), and 327 callbacks from PHP built-in functions. There were 25 callbacks to closures, which are especially challenging for static analysis, yet easily identified at runtime.

5.5.3 Evolution

Each of our three applications experienced one expansion event during the experiment. The largest event occurred in WordPress when the site administrator enabled permalinks (at request #53,310), which has the following effects on the site:

- Visitors may request any page by name. For example, the original “ugly” URL `http://ourlab/?p=18` is now also reachable as `http://ourlab/publications_page/`.
- Requests for the original URL forms `/p=` or `/page_id=` are rewritten by WordPress to the permalink URL.
- Visitors subscribing to a comment feed can use the permalink form of the feed URL (which was requested by crawlers even though comments were disabled).

A smaller data expansion occurred in DokuWiki after a change to the layout of the start page and the wiki menu bar. In GitList, a code expansion incorporated new view templates into the trusted profile as they were dynamically generated. There were no false positives during any of these expansion events, yet ZENIDS also did not add any edges to the Trusted Profile that were not specifically related to the trusted state change.

G_{wx} Safe Profile Expansion

Learning to trust the new WordPress permalinks feature was the most risky of the three expansion events, because WordPress receives every request for a URL that the webserver does not recognize. Since these are mostly malicious requests, a weak implementation of ZENIDS might mistakenly trust malicious control flow as part of the expansion event. But manual analysis confirms that all 144 newly trusted calls were strictly necessary to support the permalinks feature. In fact, ZENIDS reported 114 intrusion attempts during the expansion, including 9 attempts at known WordPress plugin exploits (CVE-2015-1579 and [38, 121, 39]), 2 invalid login attempts, 3 attempts to register new users, 6 requests for disabled features, and 36 unauthorized requests for administrator pages. Following the expansion, ZENIDS continued to report anomalies for thousands of malicious requests, many of which used a valid WordPress permalink URL form.

The GitList expansion incorporated several new Git views into the trusted profile, each having more than 100 SLOC. In DokuWiki the expansion added 145 new SLOC. We did not experience enough attacks targeting GitList or DokuWiki to make an empirical case for the safety of those expansions, but manual analysis confirms that every added call was strictly necessary for ZENIDS to trust the newly enabled features.

The Twig template engine in GitList conforms to all three characteristics of a disciplined code generator, indicating that ZENIDS successfully detected the three corresponding criteria in hook **H4** when the new views were generated.

G_{dnd} Sufficient Profile Expansion

In all three applications, no false positives occurred for requests that accessed the newly trusted features, although the features are substantially complex. For example, in WordPress the permalinks support many URL variations and a complex resolution mechanism. Although it was heavily exercised for the 9+ months during which permalinks were enabled, ZENIDS trusted all the new code paths on the basis of the 2,000 request expansion period. The new GitList views and DokuWiki menu layout also activated new URL forms with additional URL resolution, and these were fully trusted after the expansions completed.

5.5.4 Resource Efficiency

A complex tool like ZENIDS can potentially consume many different kinds of resources, including administrative effort to deploy and configure the runtime, development of the runtime and related tools, and execution time. The goal is simply to minimize these costs for the typical usage scenarios, and ideally provide a positive cost/benefit for every potential user.

$G_{\textbf{lite}}$ Runtime Performance

We evaluated the performance of ZENIDS by replaying a contiguous segment of the recorded HTTP traffic from the experiment in Section 5.5.2. To avoid bias we selected a segment having a representative frequency of trusted user requests and expansion events (though none incorporated new control flow). The web server is an Intel Xeon E3-1246 v3 with 32GB RAM and a solid state drive, configured with a LAMP stack running on Ubuntu 14.04: Apache 2.4.7, MySQL 5.5.49, and PHP 7.1.0 alpha 3.

To show that ZENIDS performs well in a real deployment scenario, we configured optimizations that would typically be used for high-traffic PHP websites. For example, our deployment enables the popular `opcache` extension, which alleviates compilation overhead by caching the compiled opcode sequences, after performing 12 low-level optimizations. We also chose the latest stable build of PHP which includes significant core optimizations such as global register allocation for the script instruction pointer and frame pointer. We compiled all PHP modules with gcc optimization flags `-O3 -march=native` to obtain peak performance for the baseline (vanilla) configuration of the PHP interpreter.

After configuring ZENIDS with the same trusted profile that we reported in Section 5.5.2, we replayed the HTTP requests synchronously to isolate the PHP interpreter's execution time from process and thread scheduling factors. Table 5.4 shows the overhead is less than 5% for all three applications.

	WordPress	GitList	DokuWiki
Runtime Overhead	4.1%	4.6%	4.5%

Table 5.4: Runtime overhead of ZENIDS vs. an optimized vanilla LAMP stack, measured as the geometric mean of 10 runs.

$G_{\textbf{dep}}$ Deployment Effort

ZENIDS can be installed in the same way as any build of PHP, though it does require the customized version of the PHP core interpreter, which includes some additional instrumentation and hooks. Using the default configuration of the IID Service for both profiling and logging, it is possible for these tasks to be completely transparent to the site administrator, similar to BlackBox.

$G_{\textbf{dev}}$ Development Effort

The cost of developing and maintaining ZENIDS is an important factor in its overall performance as a practical security tool. Although there is significant effort involved, this burden becomes progressively lighter as the user base grows, since the work only needs to be done once for each version of the PHP interpreter.

The ZENIDS PHP extension consists of 20KLOC of code, and we additionally instrumented 8 hooks for a total of 317 lines of code in the PHP interpreter source. For applications that store state in a database, ZENIDS requires a database schema to contain the $state\_taint$ that is used by hooks

H4 and **H5** to support data expansions. The schema consists of two small tables, plus one trigger per application table.

For PHP applications having an authentication scheme, the login function must be instrumented with callbacks to `set_user_level($level)`, which is provided by the ZENIDS extension as a PHP built-in function. The `$level` argument is an integer indicating the new authentication level, for example in WordPress we use the application's `role_id`. Placing the callbacks was simple in both WordPress and Doku: for each application, we inserted three callbacks in one source file immediately following a log statement indicating successful login or logout (GitList has no authentication).

5.5.5 Verifiability

The ZENIDS prototype is open-source and can be found at `http://www.github.com/uci-plrg/zen-ids`. Since the data used in our experiments contains private user information, we are not able to publicly release it. However, our repository provides instructions for conducting similar experiments in ZENIDS.

Chapter 6

Related Work

In an effort to understand the current state of software security and to improve protections for system data and resources, researchers have developed numerous attacks and defenses that push the boundaries of the security domain. Published attacks expose vulnerabilities that were not previously recognized, and show the potential risks faced by affected systems. Published defenses mitigate or even defeat these attacks with advanced strategies for eliminating known vulnerabilities or the adversary's opportunities to compromise them. But before stepping into the history of this ongoing combat, it is important to understand the motivation for this research process and consider its limitations. After all, wouldn't it be easier if we could develop quantitative measures of security that apply to all possible attacks?

Several factors make it difficult to formalize the degree of security provided by a software defense technique. To begin with, the attack surface of popular and important applications is constantly changing as development tools and techniques evolve. Within the context of a stable platform, attackers continue to discover new vulnerabilities and new approaches to exploiting them. Even if a defense is proven perfectly secure against a certain attack vector, there is no way to quantify the potential for a new attack vector to be discovered. Adding to the challenge, users have a low tolerance for overhead from a security tool—it is perceived as contributing nothing to the value of the software, serving only to preserve existing functionality. This has driven defense efforts towards heuristics or other approximations to gain a slight performance edge, instead of focusing on fundamental, axiomatic security concepts [76].

The net effect of these factors is that the security goals of the research community have become a moving target. Without a systematic approach for evaluating the effectiveness of a defense technique, research efforts have focused on the development of attacks. While it makes for a somewhat unscientific research process with transitory results, this approach has been widely accepted.

Accordingly, one of the major motivations for developing IID is that the ultimate security benefit of any given defense is unknowable under the current state of the art. Since there is no scientific certainty about whether a new control flow attack may be possible, an IID can complement existing defenses by selectively recording application state and progress during the execution of suspicious program activity. This includes control flow events and related context information such as call stacks and even heap snapshots that can be invaluable to forensic analysis and application debugging efforts alike. An IID can also assist in correcting and/or improving installed defense systems to incorporate newly discovered attack vectors into the scope of protection.

Rather than advertise the merits of IID on the basis of this philosophical argument, this chapter presents a thorough analysis of existing defense systems in both CFI (Section 6.1) and TID (Section 6.3). The goal of this analysis is to show:

- Known attacks can defeat each important CFI and TID approach.
- An IID can detect exploits that existing defenses may allow.
- The IID design supports many important features beyond exploit detection.

These sections conclude with a cost/benefit analysis of IID in comparison with existing defenses.

As mentioned in Section 3.1, the anticipated release of Intel's security extension CET could significantly change the security landscape on the x86 platform. To account for this potential future scenario, Section 6.2 revisits the defenses of Section 6.1 to show how their effectiveness may change under CET, and how IID can continue to be a valuable approach.

Section 6.4 focuses on security research specific to PHP applications, which takes the form of vulnerability detection scanners. Section 6.5 presents an orthogonal defense strategy that focuses on eliminating bugs and other unwanted behaviors from programs, and shows how IID can complement the ongoing effort to improve these important tools. Section 6.6 focuses on the usability of IID, presenting work in such areas as program comprehension and symbolic execution that may be able to reduce or eliminate the inconvenience costs of IID.

6.1 Control Flow Integrity

Introspective Intrusion Detection can provide valuable insight into unexpected program behaviors, including many that could be missed by leading CFI defenses. Sections 6.1.1 and 6.1.2 present a detailed, statistical comparison of BLACKBOX vs. CFI techniques based on equivalence classes of safe branch targets, showing that the accuracy of IID is almost always higher because it defines more equivalence classes with fewer members. Section 6.1.3 compares BLACKBOX to an orthogonal approach to CFI that focuses on protecting control flow pointers. While this can enable perfect accuracy in the set of allowed targets, it often exposes other vulnerabilities. Section 6.1.4 presents sophisticated attacks against x86 applications, including targeted attacks against that can disable various components of CFI defenses. Section 6.1.5 discusses compatibility of leading CFI tools with popular applications and operating systems, and Section 6.1.6 compares BLACKBOX performance to the CFI techniques presented here. The discussion of CFI concludes in Section 6.1.7, which summarizes the advantages and disadvantages of IID in comparison and contrast with CFI.

Throughout this comparison with CFI defense systems, it is assumed that offline procedures are free of adversarial influence, including IID profiling and the static analysis of the CFI defense. This scenario was addressed earlier in the context of Paradise (Section 2.3.1).

6.1.1 Equivalence Class CFI for x86

One approach to securing control flow is to constrain dynamic branches to a set of valid targets that is computed offline in an environment free of malicious influence. The defense does not attempt to prevent the adversary from modifying a branch target, but instead constrains the branch at runtime. This may be implemented by forcing the target to be selected from the valid set, or by detecting that a target outside the valid set is being executed and terminating the program instead. The challenge faced by these techniques is that the set of valid targets is not exactly known until runtime, at which point the adversary is already present. Computing the target sets offline requires approximation by grouping targets into *equivalence classes*, and assigning each branch to the equivalence class that most accurately represents its valid target set. Since this factor is central to the degree of security provided by these techniques, they can be called *equivalence class CFI* (EC-CFI). There are two important factors that define the effectiveness of an EC-CFI:

- The number of targets allowed at each branch, especially considering the largest instance of these sets, because conceding control of even one branch can lead to total defeat.
- Robustness to attacks directed at the components of the defense.

Since the second point is relevant to all software defense techniques, it will be covered more generally in Section 6.1.4. The goal of an EC-CFI is to make the best compromise between three competing demands:

EC1 If the equivalence class assigned to a branch is too small, the protected program will not work, because it needs to use a branch target that the defense tool has prohibited.

EC2 The larger the equivalence class for a particular branch, the more options the adversary has for manipulating that branch without triggering the defense.

EC3 Runtime overhead typically increases with the number of equivalence classes, such that providing better security often means degrading program performance more.

IID is an EC-CFI that minimizes EC2 slightly beyond the limitations of EC1. This is possible because an intrusion detector does not interfere with the program when a false positive occurs. However, if an IID goes too far in this direction, it generates too many false positives and becomes useless. So the challenge for IID is very much the same as for any other EC-CFI, but IID is distinguished from these defenses by the fact that it stands on the opposite side of EC1. On the x86 platform, IID also pushes the limits of EC3, as discussed in Section 6.1.6. This section will focus on comparison of BLACKBOX with EC-CFI techniques for x86 user-space programs, and Section 6.1.2 will more generally compare IID with EC-CFI techniques for other platforms.

The research question to address here is how to concretely compare the accuracy of IID detection vs. the accuracy of EC-CFI defense. As discussed in the CFI survey by Burow et al. [19], it can be challenging to compare one EC-CFI to another because the equivalence classes can be assigned in fundamentally different ways. For example, a modular EC-CFI called MCFI [123] uses call graph analysis to assign an equivalence class to every return instruction, whereas BLACKBOX uses a dynamic shadow stack that only allows the one valid return target (for the case of soft context switches, see Section 3.3.2). The `switch` statement in C and C++ programs creates similar confusion in the reporting of metrics about an EC-CFI. Approaches like MCFI typically group all the defined `case` labels into one equivalence class and omit the `switch` from reported metrics about equivalence classes. But BLACKBOX only trusts the `case` labels that were observed during profiling, and furthermore it does not distinguish between a `switch` statement and an intra-module call. The net effect of these seemingly minor points is that it can be very difficult to make an apples-to-apples comparison of accuracy between any two implementations of EC-CFI. One way to make consistency a priority is divide the comparison into five categories that can, for the most part, be addressed in a homogeneous way across EC-CFI techniques: virtual calls, indirect intra-module calls, cross-module calls, indirect jumps, and returns. It is not clear how to summarize the results of these five comparisons, but it turns out not to matter because BLACKBOX significantly undercuts all the EC-CFI competition for accuracy in almost every category—though at the cost of performance and inconvenience.

Virtual Calls

A specialized form of EC-CFI focuses exclusively on protecting the virtual dispatch tables in x86 binaries compiled from C++ source. The traditional mechanism for implementing virtual calls in x86 machine code is vulnerable to manipulation, for example a counterfeit object attack masquerades an attacker-controlled buffer as a compiled C++ object filled with malicious virtual function pointers [149]. The type system of the C++ source precludes the majority of these artificial targets, but is not enforced by the traditional dispatch mechanism for performance reasons. The goal of virtual dispatch CFI (VD-CFI) is to constrain these tables according to the C++ type system. This translates directly into EC1 and EC2, because the net effect is to constrain the set of functions that can be reached through the dispatch table.

Leading VD-CFI tools vary slightly in the details of the equivalence classes, which are evidently sufficient to prevent COOP attacks, and the runtime overhead, which is typically under 3%. SafeDispatch [87] extends Clang++/LLVM to insert a set membership test at every virtual call site, ensuring that the function selected from the vtable is coherent with the class referenced at the call site. Since set membership is not easy to implement efficiently, the same authors upgrade the work in VTI [12] by ordering vtables according to a pre-order traversal of the class hierarchy, which allows the same test to be implemented as a simple range check. VTV [171] extends LLVM with a similar implementation to SafeDispatch, but with support for dynamic linking of modules. It may not be cost effective for VTV to implement the pre-order layout of VTI because its runtime component may adjust the set of allowed vtables anytime a module is loaded or unloaded.

To avoid dependence on source code, T-VIP [59] and VTint [194] employ binary analysis and binary rewriting to identify virtual method calls, identify the relevant class hierarchy, and instrument the call with A dynamic approach based on the binary translator Pin [105] reports similar effectiveness in the size and relevance of equivalence classes, but its runtime has even higher overhead than BLACKBOX [140].

The comparison between VD-CFI and IID is relatively easy to make, even though BLACKBOX does not distinguish a virtual dispatch call from any other call. There are two important distinctions:

- As with all forward branches, BLACKBOX defines one equivalence class per call site. In contrast, a VD-CFI defines one equivalence class per virtual dispatch table, which may be referenced by many call sites.
- The equivalence class defined by VD-CFI always includes the complete set of functions corresponding to some slice of the C++ class hierarchy, whereas in BLACKBOX the Trusted Profile only contains the specific call edges that were observed during profiling.

The net effect of these two differences is that BLACKBOX always has an equivalence class the same size or smaller than is generated by a VD-CFI. Intuitively, the Trusted Profile may contain a different set of targets for two call sites that access the same virtual dispatch table. More commonly, the Trusted Profile contains no targets because the call site itself is not trusted, whereas the offline approach of VD-CFI must trust some non-empty set of targets at every call site. An implicit limitation of VD-CFI is that it only applies to C++ programs that make extensive use of the class hierarchy. In contrast, BLACKBOX applies the same Trusted Profile constraints to a call generated from any source, including hand-coded assembly and code generated by JIT engines for script interpreters. VD-CFI has the advantage of convenience and performance, rarely exceeding 3% overhead on standard benchmarks and requiring no end-user involvement.

Intra-Module Indirect Calls

Although other, more conventional forms of EC-CFI do not typically modify virtual dispatch tables, the static analysis to constrain virtual call sites often draws on the same logic about the C++ type system, resulting in similar equivalence classes. An EC-CFI will typically instrument the call site with a check against the set of pre-determined valid targets. The original CFI implementation by Abadi et al. [1] proposed a single equivalence class for every function pointer, allowing them to reach any function whose address is taken. Investigating simplifications for performance optimization, binCFI [196] and CCFIR [195] assign all function entry points to a single universal class, which is much larger and therefore much less secure. Both IFCC [171] and MCFI [123] group all functions having a signature matching the function pointer, though IFCC optionally supports a single universal class as well. For any given function pointer, the set of targets in the Trusted Profile will always be equal to or smaller than any of these static equivalence classes.

Three approaches improve upon static EC-CFI by putting a dynamic spin on both virtual calls and function pointers, with varied success.

- π**CFI** maintains a statically computed CFG in the background but initially disables all control flow paths. As execution proceeds, corresponding paths are *activated* by path entry points. This prevents an attacker from using one half of a control flow construct without ever using the other half, for example a return edge without the matching call. But since the activation code is inlined at call sites, this approach never prevents an attacker from choosing any call target in the statically defined equivalence class. So while some constraints in πCFI are dynamically activated, equivalence classes for calls are effectively defined by the offline static analysis.

- **PathArmor** introduces context sensitivity to EC-CFI by delaying the static analysis until the invocation of *security-sensitive functions* (SSF), which are manually defined. Instead of statically calculating sets of allowed targets, PathArmor intervenes at an SSF and walks the LBR register to evaluate whether the recent control flow edges can feasibly occur in a CFG corresponding to the current runtime context. For the last 16 branches of any execution trace leading up to an SSF, this approach can result in a smaller equivalence class than the Trusted Profile, which is context insensitive. On the other hand, BLACKBOX employs a dynamic Stack Spy that can raise suspicion at system calls where PathArmor may be blind. For example, an evasive attack can fill the 16-slot LBR with an innocuous call to a library function en route to the SSF, whereas Stack Spy retains suspicion for any number of branches until the suspicious stack frame returns. Lacking a concrete basis for distinguishing a winner, it can at least be concluded that IID may provide valuable insight into important attack scenarios that are challenging for PathArmor.

- **OpaqueCFI** combines two pervasive problems in EC-CFI into a solution. The inevitable consequence of checking equivalence classes is runtime overhead, leading many EC-CFI techniques to reduce the number of classes and arrange the data representation into aligned blocks for efficient range checking [196, 195]. But this weakens CFI enforcement, not only because the attacker has more options at any given branch, but also because the attacker can easily figure out which options are available. OpaqueCFI concedes a lesser degree of approximation, then hides the slack from adversarial discovery by waiting until program startup to randomly shuffle the aligned blocks. Even if the adversary is able to leak the CFI data structures at runtime, that information will likely be of little value on subsequent executions. So the Trusted Profile may have the smaller equivalence classes, but if the adversary gains access to it (perhaps through some unprotected program), BlackBox may become more vulnerable to evasion than OpaqueCFI.

Three approaches avoid modifying the protected program by enforcing CFI constraints through a runtime component.

- **Lockdown** [131] extracts the CFG at runtime, mostly during module load, though applying some just-in-time heuristics for constructs such as dynamically registered callbacks. The accuracy of the CFG largely depends on optional symbol tables in the binary, which identify static (local) functions and imported functions. Without symbols, Lockdown falls back to the same single-class constraints as binCFI and CCFIR. Runtime overhead is more than 2× higher than BlackBox on the SPEC CPU 2006 benchmarks, since Lockdown is based on the slower binary translator libdetox [132] and performs more analysis at runtime. An additional advantage of BlackBox is that it never relies on symbols, instead modeling the Trusted Profile with module identifiers and instruction offsets relative to the top of the module image in memory.

- **CFIGuard** [192] relies on a conventional static CFG but enforces it in a runtime that does not require instrumenting the protected program. To observe application control flow, CFIGuard relies on a kernel modification to interrupt the program after a fixed window of instructions and then reads the LBR. Intra-module indirect calls are constrained by function signature, which is substantially more effective for small modules than, for example, the massive 68MB `chrome_child.dll` that implements most of Google's browser. Not only does BLACKBOX constrain indirect branches more tightly to the set of dynamically observed targets, most branches are constrained to zero targets, allow much high visibility into unexpected program behavior. The inconvenience cost of CFIGuard presents an interesting opportunity for BLACKBOX because, if widely adopted, it becomes nearly free. An IID with very low performance overhead could be constructed by simply plugging a Trusted Profile into CFIGuard.

- **Griffin** [60] leverages Intel's new Processor Trace (PT) feature to apply existing EC-CFI policies from the kernel asynchronously. While not nearly as efficient as CFIGuard, it is able to enforce the context sensitive policy of PathArmor for forward edges in combination with a full shadow stack. The asynchronous processing of trace data can effectively mask the overhead by using cores that are often idle in today's highly parallel processors. The inconvenience costs are similar to BLACKBOX, requiring a kernel module and incurring 9.5% overhead on the SPEC CPU 2006 benchmarks.

Indirect Jumps

Most EC-CFI tools, whether based on analysis of source code or a compiled binary, will assign all the `case` labels to a single equivalence class of the containing `switch` statement. Although πCFI lazily activates the branches reachable from a `case` label, it allows any `case` label to be activated, resulting in the same single equivalence class for each `switch`. BLACKBOX does not distinguish between an indirect jump and an indirect call, and for a `switch` statement that was never observed during profiling, BLACKBOX does not trust any of its `case` labels. According to Burow's survey, the only current CFI technique to support a flow-sensitive policy is KernelCFI [61] (see Section 6.1.2).

Tail calls are implemented in x86 as an indirect jump, since by definition there is no need to adjust the stack pointer. Most CFI techniques will protect this branch as if it were a call instruction, assigning the corresponding constraints. One exceptional case is Lockdown, which only assigns one equivalence class to all tail calls within a particular module—and if symbols are missing, that class includes all addresses in a valid section of the module image. For CFI that statically computes backward edges, the set of return targets following a tail call is much more difficult to determine, leading to larger equivalence classes. For example, both πCFI and MCFI recommend disabling tail call optimizations to avoid losing precision in the return constraints. BLACKBOX does not distinguish an indirect call from an indirect jump, so its accuracy in comparison with other EC-CFI remains the same for this special forward edge. The BLACKBOX shadow stack also maintains its accuracy for tail call unwinds.

Returns

Researchers unanimously agree that a shadow stack provides optimal protection of return instructions, but many EC-CFI use an equivalence class for performance reasons. According to Burow's survey, the only EC-CFI for x86 to implement a shadow stack are Abadi's original CFI and Lockdown. PathArmor is able to validate return addresses up to the precision of a shadow stack using the LBR. Other CFI approaches limit return targets to a call-preceded instruction constrained to some degree of approximation, including call graph sensitivity and signature sensitivity. An optional constraint filters out functions whose address is never taken within the module. Several approaches simplify the constraints to all call-preceded instructions everywhere, including binCFI and CCFIR, but this is consistently vulnerable to gadget synthesis attacks [43, 63]. An advanced attack called Control Flow Bending (CFB) [21] claims it can consistently control return instructions against any CFI without a fully dynamic shadow stack, but that is the only way it can gain arbitrary code execution in the protected process (it does not mention whether the PathArmor implementation qualifies). Among the EC-CFI for x86 that implement a shadow stack—and are therefore able to detect (or defeat) an RCE attack via CFB—BLACKBOX has the best performance on the SPEC CPU class of benchmarks by more than 5%. This performance comparison is somewhat imprecise because Original CFI uses version 2000 while BLACKBOX and Lockdown use version 2006 of the benchmarks, not to mention that Original CFI was tested a full decade earlier on significantly less advanced hardware (Pentium 4 vs. Xeon E3 for BLACKBOX and Core i7 for Lockdown).

Cross-Module Calls

Though prevalent in today's popular software (Section 3.6.4), cross-module control flow is perhaps the most neglected in leading CFI research. The just-in-time static analysis of PathArmor makes it the best equipped to assign a tight equivalence class to cross-module calls, but its reliance on the LBR makes it vulnerable to evasion by history flushing. The author explains this does not seem to be a problem in the context of the SPEC CPU 2006 benchmarks, but does not mention the deliberate intention of those benchmarks to minimize library calls. Of the VD-CFI, only VTV incorporates a runtime component to update its model of active class hierarchies as modules are loaded and unloaded. For instance, SafeDispatch simply raises a false positive on a cross-module virtual dispatch. Lockdown separates constraints by module and leaves the door open across modules by defining just one equivalence class that includes all the module's functions. CFIGuard, MCFI and πCFI are slightly better, multiplexing all valid module exit paths with all valid module entry paths. Since IFCC defines an equivalence class per function signature, its relatively loose constraints remain the same across modules. The same can be said for binCFI and CCFIR, except that their equivalence classes are much larger. Original CFI does not address modularity. Opaque CFI guards the IAT to allow normal native calls to imported functions, and specially protects the library pointer for dynamically bound calls to prevent tampering. This may be the most effective approach, though does not protect against vtable tampering or other means of changing a callee away from the call site. Where BLACKBOX already has the smaller equivalence classes for indirect branches in general—since the Trusted Profile is built purely from observed executions—it does not relax its constraints at module boundaries like so many other CFI approaches. Even for cross-module callbacks the Trusted Profile contains just one target at the majority of callback sites, after weeks of usage in complex applications such as Microsoft Office. While cross-module branches may be more challenging for the adversary to attack because of ASLR and uncertainty about installed library versions, IID is prepared to detect and report manipulation of these branches.

RockJIT

The unique form factor of this EC-CFI makes it more easily discussed separately from the preceding approaches, though it does apply to x86 user-space applications. As its name suggests, RockJIT [124] focuses on securing the dynamically generated code produced by a script interpreter's JIT compiler, along with the statically compiled code of the JIT compiler itself. The authors of MCFI developed this approach to incorporate the majority of DGC into the domain of modular CFI. Though it is presented as a generic approach independently of any particular JIT engine, it does not offer a protective mechanism for *ad hoc* code generators such as are manually implemented within the Microsoft Office module `mso.dll`. To secure the code generation process, RockJIT imposes constraints on the JIT engine's use of memory-related system calls, and double-maps the memory holding generated code to separate writable and executable permissions for any given access address. It also applies MCFI to the JIT engine, protecting it from malicious scripts attempting to cross the semantic gap. Two equivalence classes are applied to forward edges in the generated code, one for direct branches (which are manipulable during code generation), and another for indirect branches. The JIT engine is instrumented (800 lines for Chrome V8) with a verification pass before emitting generated code, which evaluates the equivalence classes along with other properties such as a machine code vocabulary that is JIT engine is known to use (manually defined). The equivalence class for indirect branches is additionally checked during traversal of each dynamically generated indirect branch.

As presented in Section 3.4, BLACKBOX does not maintain the details of JIT code in the Trusted Profile, but instead focuses on the interface between the DGC and the system, including calls to the JIT engine and system calls. For disciplined code generators such as Chrome V8 and Mozilla Ion, which never generate direct system calls and never call modules other than the JIT engine, this enables reporting of any DGC exit branch that does not conform to the trusted JIT engine API (or that targets any other module or system call). Special edges are also added to the Trusted Profile for trusting the code generation process of a JIT engine (or any other code generator), which enables reporting of code generation on call stacks that are not trusted for that purpose. This is in some cases stronger than RockJIT, and some cases weaker. For example:

- A JIT spray attack using only forward edges to take control can use any sequence of calls to the JIT engine API without detection by BLACKBOX, yet would in many cases violate the RockJIT equivalence class during the takeover.
- A CFB attack on either the DGC or the JIT engine can gain arbitrary code execution against RockJIT, but cannot escape detection by the BLACKBOX shadow stack.

BLACKBOX is additionally able to monitor smaller fragments of DGC, such as those produced by the Microsoft Managed Runtime, with a context-sensitive Trusted Profile, along with its DGC memory permissions.

The inconvenience cost of BLACKBOX is higher considering the performance difference, roughly 2.5× vs. just 14% on the Octane JavaScript benchmarks with Chrome V8. But BLACKBOX can be used on a COTS browser while RockJIT requires instrumenting the browser and applying MCFI at runtime.

6.1.2 Equivalence Class CFI for Other Platforms

Several CFI approaches have been developed for platforms where there is currently no IID implementation. Lacking a reliable way to predict the effect of various snares that may occur on these platforms, this section proceeds to compare IID with each of these tools in more general terms.

CFI for Kernels

Some challenges faced by CFI implementations are simplified in the context of kernels, for example modules are most often statically linked, and the use of function pointer variables often follows reliable patterns. An CFI called KernelCFI [61] takes advantage of these properties to both simplify and optimize its EC-CFI protection of the Linux kernel, MINIX and BitVisor. To compute the equivalence classes for function pointers, KernelCFI applies taint tracking to function addresses, which is tractable under the assumption that pointer arithmetic and aliasing with data pointers are prohibited. Computing return targets is complicated by hand coded assembly which in some cases includes nested functions and non-canonical forms of tail calls. Since linearity of execution in a kernel is fragmented by interrupts, the runtime component of KernelCFI is much more complex than many user-space approaches. Care must be taken to protect event handlers and internal control tables. The implementation is manually instrumented in the kernel and also manually optimized, for example replacing some indirect branches with direct branches adhering to singleton CFI constraints. Given the amount of manual attention in this approach, it is difficult to compare the size of equivalence classes or the effective security overall. Average runtime overhead ranges from 2-12%.

Secure Virtual Architecture (SVA) [37] pioneered the effort to integrate EC-CFI throughout the complexities of kernel operation. SVA employs a typesafe language that adheres to the SAFECode [46] principles for enforcing safety in unmodified C programs. Equivalence classes are enforced by grouping all pointers for a class into a memory pool and instrumenting membership checks as kernel modules are compiled in LLVM. A points-to analysis is performed at the SVA language level to identify the minimal statically determinable equivalence classes, and other kinds of errors are strictly forbidden by the SVA language, resulting in relatively strong security in comparison to user-space CFI approaches. Performance evaluation of a Linux port to SVA demonstrates overheads ranging from 2% for CPU-bound operations to over 4× for I/O bound operations or for tasks that involve frequent calls to complex system functions like fork/exec.

KCoFI [36] extends SVA with a compiler that optimizes many of the integrity checks and increasing the size of equivalence classes, but maintains other security properties (as demonstrated with lengthy formal proofs). Overheads still exceed 2× for I/O bound workloads, but usability is significantly better than SVA. HyperSafe [181] applies similarly large equivalence classes in the context of a self-protecting hypervisor at overheads under 5%, but does not implement full coverage of kernel events or protection of internal tables, making it vulnerable to ret2usr attacks and manipulation of its event handling configuration.

Given the approach to calculating equivalence classes (and the lack of concrete experiments), it seems evident that an IID would generate a Trusted Profile with equivalence classes that are much smaller than any of these techniques. The integration could be done with a kernel mode version of DynamoRIO called drk[1], or perhaps by using the interrupt-based approach from CFIGuard with directly instrumented handling of other factors similar to KernelCFI. Performance of BLACKBOX is in the same range, suggesting an IID based on drk might be the more convenient approach. For computing operations that are highly concerned about advanced persistent threats, a kernel IID could be an essential component in the defense infrastructure.

[1] Available at `https://github.com/DynamoRIO/drk`

CFI for iOS

Control-Flow Restrictor [136] brings EC-CFI to the iOS platform by extending LLVM to extract a static CFG and insert runtime checks during compilation. The developer can influence equivalence classes for forward edges using annotations, but there appears to be only one equivalence class for returns. One advantage on iOS is that all modules must be statically linked, making it possible to at least apply a return-specific equivalence class throughout all linked libraries. Average reported overhead is just under 10%.

In another approach to EC-CFI on iOS, MoCFI [42] applies directly to compiled programs and derives the equivalence classes from binary analysis. A single equivalence class is generated for forward indirect calls having no statically computable target, and returns are checked against a full dynamic shadow stack. Instrumentation is applied at load time to avoid complications with application signing. Performance is reported piecewise without summarizing an average, but can range as high as 5× and appears to be high in general.

An IID for iOS would operate on much smaller equivalence classes, considering the conventional static analysis applied in both cases, and especially the absence of a shadow stack in Control-Flow Restrictor. The compatibility of a virtual machine like DynamoRIO with iOS seems questionable, so an approach similar to these defenses is likely to be more effective in practice. Given the performance numbers in the context of x86, it appears that a conventional implementation of IID could be competitive. An IID for Android may have even greater potential, given the presence of the Android Runtime and corresponding opportunities for high performance introspection.

CFI for Other environments

MIP [122] improves security for inline reference monitors by applying EC-CFI with similar equivalence classes and implementation structure to binCFI and CCFIR. Performance overhead is under 10%, but the value of this approach in today's threat model is unclear. A hardware-assisted approach to EC-CFI on embedded platforms defines only one equivalence class for indirect calls, but enforces a separate equivalence class at each return instruction based on the call graph. Other heuristics are applied to detect attacks, but these are similar in design and weakness to several approaches on the x86 platform that have been effectively defeated (kBouncer [129], ROPecker [28] and Smashing the Gadgets [128]). The enforcement of CFI constraints leverages special hardware instructions that associate equivalence classes with the execution of branch instructions on the basis of a finite state automaton. No implementation or performance results are reported in the publication, though Burow's survey indicates high performance for this otherwise unreferenced approach. In any case, the value of such large equivalence classes and heuristics remains uncertain in today's environment of highly technical adversaries.

6.1.3 Pointer Protection CFI

Several techniques focus on protecting control flow pointers from abuse, without attempting to enforce any pre-determined set of constraints on the control flow taken at runtime. Similar to IID, the conceptual simplicity of pointer protection CFI (PP-CFI) makes it resilient to new forms of attacks. But the implementation can often be complicated because protecting pointers involves substantial changes to the fundamental execution of the program. Where EC-CFI simply observes the target of an indirect branch, PP-CFI changes the low-level mechanism for accessing pointers, creating effects that carry into many different aspects of the program and possibly even the operating system. Attempts to simplify these effects can lead to incompatibility with programs, unanticipated security vulnerabilities, and can even expose the PP-CFI to targeted attacks.

Cryptographic CFI [107] encrypts the target of every indirect control transfer and stores the key in a dedicated register. If the adversary manipulates a pointer in memory, the decryption will produce a random address that will most likely cause the program to crash. This approach is compatible with all indirect control transfers generated by the compiler, but if the program overwrites pointers in an unconventional way, it will crash. Dynamically generated code is not protected unless the code generator specially implements pointer encryption. Performance is the main challenge for this approach, averaging 45% on SPEC CPU 2006 with commodity hardware that provides encryption instructions (otherwise it will be orders of magnitude worse). Specialized hardware support can dramatically improve performance and may become available in some commodity processors.

Code Pointer Integrity [97] places pointers to code in a separate protected heap. This has limited compatibility with callbacks, requiring both caller and callee to be instrumented with CPI. It will also crash the program during a soft context switch because the "return location" in the protected heap will not reflect the stack pivot. While these issues are neither mentioned nor addressed by the research authors, both could be resolved in a commercial implementation. Two protection modes are offered, the more significant exceeding 8% overhead on SPEC CPU 2006.

Control-Data Isolation [4] extends LLVM to rewrite all indirect branches as direct branches, requiring whole-program compilation to incorporate cross-module control transfers. This approach does not address callbacks, soft context switches or DGC, and will generate crashing executables in those cases. Only a subset of the SPEC benchmarks and a few small server programs are tested. While there is little question that indirect control attacks are eliminated by the PitBull compiler, it only supports a very small subset of real programs.

6.1.4 RCE Exploits vs. x86 Compiled Binaries

Dynamic approaches to CFI such as kBouncer [129], ROPecker [28] and others [128] offer broad coverage of control transfers while minimizing inconvenience costs. But an attack can synthesize gadgets that do not conform to the tool's definition of a gadget, making the takeover transparent [64, 22]. Since these defenses focuses strictly on identifying gadgets used in adversarial takeover, none of them is able to detect any part of a malicious payload.

The coarse-grained CFI approaches binCFI and CCFIR are not vulnerable to padded gadgets because they continuously apply constraints to control transfers. But to reduce overhead, these techniques equivocate all targets within a particular category. Both publications assert that the potential for a reuse attack that maintains these categories is too low, but two recent attacks prove otherwise [43, 63]. Once the exploit has gained control of an application thread, these techniques have no means of constraining execution of the payload.

StackDefiler

Stack-spilled registers can expose the data structures of a CFI implementation, making it vulnerable to direct manipulation via typical stack corruption [30]. Of the techniques listed below BLACKBOX in Table 6.1, only Cryptographic CFI is immune to this attack because it stores the encryption key in a dedicated register that is never spilled to the stack. But an auxiliary component of StackDefiler manipulates the return address from any system call invoked via the x86 instruction `sysenter`, and since this return address is managed by the operating system, the Cryptographic CFI compiler will not be able to instrument it. This will allow the attacker to use any `sysenter` instruction as a gadget under Cryptographic CFI.

BLACKBOX is immune to the stack-spilled register attack—its dynamic CFI checks are performed by a hand-coded assembly routine that only spills application values to the stack, and immediately replaces the register values with the spilled application values after the check. The `sysenter` instruction is rewritten by all x86 builds of DynamoRIO into an `int 3`, making all tools based on DynamoRIO immune to the corruption of the `sysenter` return address. Even without this convenient protection, the deep introspection of DynamoRIO includes system call arguments, allowing BLACKBOX to keep the correct `sysenter` return address on the shadow stack.

Another auxiliary component of StackDefiler leaks the address of the shadow stack using typical side channels that are known to be effective against information hiding. A variation targets a shadow stack residing in thread-local storage (TLS) by exposing stack-spilled registers, similar to the main attack vector. The authors claim that this variation was not implemented because no TLS-based shadow stack was publicly available at the time of publication, although BlackBox was in development simultaneously at the same university. The attack will fail against BlackBox because shadow stack access is implemented in hand-coded assembly that never spills the shadow stack location to the application stack—the temporary registers used for shadow stack access are stolen immediately prior to each stack operation and subsequently repopulated with the application's register values.

Control Flow Bending and Control Jujutsu

Until the publication of these two exploits, the research community had mistakenly believed that if a program's execution fully adhered to a perfectly defined static CFG, that execution would necessarily exclude any possible RCE exploit. In a Control Flow Bending (CFB) [21] attack, the adversary locates a *CFB gadget function* that is (a) capable of overwriting its own return address, and (b) called by a variety of useful functions. Since the calls are effectively aliased in the CFG, the CFB gadget allows any of those call sites to be selected as an alternative return target. Similarly, in a Control Jujutsu [54] attack, the adversary first identifies an application function that dynamically forks a process by name, then redirects a forward pointer that can legitimately target the *fork gadget*, while modifying the arguments to name a malicious process. Fork gadgets may not be available in all applications, but were found in Nginx 1.7.11 and Apache 2.4.12.

Of the defense techniques listed in Table 6.1, only CPI and BlackBox are able to reliably defeat these two attacks. The case for CPI is complicated by the fact that the authors entirely neglected support for legitimate modification of values on the safe stack. But commercial implementations of CPI having compiler support are immune because they will only facilitate runtime modification of safe stack pointers by program constructs that were intended for that purpose. Since CFB and Jujutsu both use a crafted mechanism for modifying a protected pointer, the application will not propagate that modification to the safe stack, and the CPI security response will be triggered (presumably it terminates the process).

The best-case scenario for BLACKBOX is that the modified branch target does not exist in the Trusted Profile, because the application was not normally used that way. But if the adversary does find a trusted fork gadget or CFB gadget, BLACKBOX falls back to detecting the payload, as with any other exploit. There are two special cases to consider:

1. An auxiliary component of CFB can implement Turing-complete computation within a single function, for example by generating a self-modifying format string to `printf()` and artificially looping via manipulation of the index into the format string. In the case that a CFB attack leverages this capability to manipulate existing system calls in the application, the attack is transparent to BLACKBOX. But since this is a data-only attack, it is out of scope for BLACKBOX—it can be better addressed by defense techniques that focus on data-only attacks. Other CFB payloads will be detected by BLACKBOX.
2. The Jujutsu fork gadget has the effect of moving the entire payload out of process, making it transparent to BLACKBOX. But the Trusted Profile contains an edge for every process fork, tagged with the full path of the spawned executable, allowing BLACKBOX to report a high-suspicion anomaly when a Jujutsu exploit successfully forks a malicious process.

Missing the Point(er)

Code Pointer Integrity has proven effective enough to be implemented in major commercial products, but requires hardware support to hide the safe stack. On x86-64 and ARM, isolation can only be enforced by information hiding, making it vulnerable to an information leak attack. CPI leaves data pointers in the original application stack, allowing this attack to leverage a data pointer for a timing side-channel that can reveal the location of the safe stack without causing a crash. This approach is not prohibitively time-consuming—it is able to take full control of a CPI-protected deployment of Nginx in just 98 hours [53]. BLACKBOX is complementary to CPI and would be just as effective in detecting this attack, whether CPI were in effect or not.

The Devil is in the Constants

The case study in Section 3.4.2 shows how BLACKBOX can detect this JIT injection attack even if the monitored application provides no protection from adversarial JavaScript constants [5]. None of the other defenses in Table 6.1 is applicable to dynamically generated code, making them unable to protect the application from this attack. RockJit [124] is a coarse-grained CFI implementation specific to the code generated by JIT engines, but suffers from the same vulnerabilities as binCFI and CCFIR and can be similarly compromised by this code injection. JIT diversification tool Librando [77] somewhat inadvertently defeats this exploit by revoking executable permission to all memory pages containing generated code, and linking a runtime-diversified copy of the code in the style of a binary translator. While Librando does not diversify or obfuscate the malicious constants in any way, the fact of disabling execution on the original JIT-generated code pages effectively disables the exploit gadgets.

6.1.5 Application Compatibility

Many of the defense techniques listed under BLACKBOX in Table 6.1 have been shown to be easily defeated, and others suffer major compatibility problems with ordinary applications. Approaches [196, 195, 129, 28, 128] offer coverage of many important control flow constructs at low overhead, but have been defeated by gadget synthesis attacks [43, 63]. It is also possible to defeat [97] on the 32-bit x86 platform. Control Data Isolation [4] appears to eliminate the problem of control flow hijacking entirely, by simply flattening all indirect branches, but this approach is incompatible with the vast majority of today's popular applications. For example, no native Windows desktop application can be protected by this tool—it will either fail to compile or crash at startup. Both CDI and binCFI will evidently crash on a cross-module callback.

Cryptographic CFI [107] requires hardware cryptographic extensions and a reserved register, which is highly impractical for the x86 platform where general registers are already far too few. It is also unclear how Cryptographic CFI can protect a callback pointer passed to a library that was not compiled with its instrumentation. CPI can only protect a callback pointer if both the caller and callee have its instrumentation. Many approaches in the table will evidently encounter a false positive ($\times_{fp}$) at a cross-module callback. CFIGuard mentions no support for complex forward control flow such as soft context switches. Others assign only the most basic constraints such as byte alignment [114, 1]. Perhaps the best case scenario is the VD-CFI approaches, but these only affect C++ pointers distributed from virtual dispatch tables. Performance is a significant limitation for [1, 131, 59, 140]. This leaves diversification [101, 78] as the only practical approach, though it has met with some resistance in practice due to complications with integration into the software production toolchain. Even if these defense techniques were unbreakable, only diversification and kBouncer would match BLACKBOX in coverage of control flow constructs.

6.1.6 Performance

For a security tool to be considered for deployment in many commercial products, the overhead can be no more than 5%. Most of the CFI tools presented here are below that mark or near enough to reach it with conventional engineering effort. Among the more effective techniques, Crypto CFI is perhaps the exception, though it has recently received significant attention from hardware manufacturers and would function efficiently with dedicated CPU support. RockJIT is limited considering the intense pressure for performance in today's web browsers, but may see substantial

		Dependencies		Defense Coverage				
Defense	Overhead	Requires Source	Static Analysis of Indirects	Backward Edges	Callbacks	Soft Context Switch	DGC	Cherry-pick Benchmarks
BlackBox	14.7%			✓	✓	✓	✓	
binCFI [196]	8%		bin	$\checkmark_{cp}$	×!	$\checkmark_{cp}$	×!	●
CCFIR [195]	5%		bin	$\checkmark_{cp}$	×	$\times_{fp}$	×	
CFIGuard [192]	2.9%		bin	$\checkmark_{cp}$	×	$\times_{fp}$	×	●
Control-Data Iso [4]	10%	●	src	✓	×!	×!	×!	●
Context CFI [174]	3-8.5%			✓	✓	$\times_{fp}$	×[a]	●
Crypto CFI [107]	45%	●		✓	✓	✓	×	●
Prob. NOP Insertion [78]	< 1%	●		✓	✓	✓	×[b]	
Forward Edge CFI [171]	-[d]	●	src	×	×	×	×	●
kBouncer [129]	1%	●		✓	✓	✓	✓	●
Lockdown [131]	32.5%		bin	✓	×	$\times_{fp}$	×!	
OpaqueCFI [114]	< 5%		bin[c]	✓	×	×	×	●
OriginalCFI [1]	21%		bin	✓	×	×	×	
Pointer Integrity [97]	2-9%	●	src	✓	✓[e]	×!	×	
ROPecker [28]	2.6%		bin	✓	✓	$\times_{fp}$	×	
SafeDispatch [87]	2.1%[f]	●	src	×	×	×	×	
Smash Gadgets [128]	0%			✓	✓	✓	×	
T-VIP [59]	2-103.5%		bin	×	×	×	×	
vfGuard [140]	18.3%		bin	×	×	×	×	●
VTI [12]	1%		bin	×	×	×	×	
VTint [194]	2%	●		×	×	×	×	

Table 6.1: Compatibility issues and other inconvenience factors in leading control flow defenses for x86 user-space programs.

$\checkmark_{cp}$ Equivocates all call-preceded code addresses in some or all cases.

$\times_{fp}$ Raises a false positive on every occurrence.

×! Crashes the program on any occurrence.

[a] Publication makes no mention of DGC.

[b] Another work by the same author applies a similar technique to DGC [77].

[c] Optional source-based analysis improves accuracy.

[d] Overhead varies according to the selected security options.

[e] Requires both the caller and the callee to be instrumented.

[f] Requires whole-program profile-guided optimizations and analysis.

improvements in a commercial implementation. Kernel applications of CFI also lag behind their user-space counterparts, but it will be interesting to see if facilities such as the LBR can be used to reduce these overheads.

At 14.7% overhead, BLACKBOX appears unable to make the 5% threshold, especially considering that its underlying framework DynamoRIO has at best 11% overhead on the popular SPEC CPU 2006 benchmark suite. But these statistics do not necessarily represent a fair evaluation of a tool's usability. BLACKBOX focuses on desktop applications for the x86 platform, yet the SPEC CPU 2006 suite does not include any typical components of desktop applications. As discussed in Section 3.6.3, the user experience of BLACKBOX in this scenario is similar to native execution except while the application loads modules or generates and executes large amounts of code. There are also many important use cases for IID that do not require continuous activation throughout the group of monitored users. Even where sampling is not sufficiently secure, selective monitoring for longer periods can minimize the impact of overheads. Server applications also perform well under BLACKBOX, for example Section 3.6.3 reports a small speedup for a benchmark of the IIS web server. Conversely, the performance metrics reported by many CFI authors are shown to be substantially lower than were observed during an independent evaluation in Burow's survey. Given the lack of support for many important control flow constructs, and weak support for common factors such as cross-module indirect branches, it would be interesting to see how these tools perform after completing their implementation.

There may also be substantial opportunities to improve IID performance on the x86 and similar platforms. For example, an expansion of the LBR and better OS support for hardware-based monitor could make an out-of-process IID relatively easy to implement, robust to adversarial detection and manipulation, less intrusive to low-level program behavior, and highly competitive in performance. Intel PT may represent a similar opportunity, though less appealing from a performance standpoint. Extensive work in the area of virtual machine introspection (VMI) [58] may also hold potential for a less intrusive and better performing IID. Considering the number of years that commodity hardware has been effectively treading water in terms of major performance advances, it would not be surprising to see revolutionary innovations in the near future that could change the performance picture for IID on hardware. Given the cost and complication raised by security problems on existing platforms, hardware vendors may also focus more resources towards introspection and program awareness than in past designs. While none of these possibilities can be anticipated with certainty, they lend credibility to the aspiration that IID is not far from becoming a minimally intrusive and highly efficient means of monitoring vulnerable programs.

6.1.7 Summarizing IID in the field of CFI

In the context of these diverse approaches to enforcing CFI constraints on end-user programs and kernels, the advantages and limitations of IID stand out in distinct contrast. On the matter of effective security, while IID can only prevent an attack if it has been manually blacklisted, but it can detect and report a far larger set of control flow attacks than existing EC-CFI defenses. The Trusted Profile defines smaller equivalence classes in nearly all forward indirect branch scenarios, and the shadow stack is fundamentally more accurate than any equivalence class scheme. In addition to precise handling of cross-module indirect branches, an IID can detect anomalies in dynamically generated code at multiple levels, including the code generation process and in many cases context-sensitive sequences of generated instructions. Where many research implementations suffer from incompatibility with current programs and operating systems, making prototypes unusable in practice, IID has been significantly exercised in real-world deployments for both PHP applications and x86 desktop and server applications. There are a few special cases where an EC-CFI approach can prevent an exploit that IID may not detect, for example a Control Jujutsu attack that is transparent to a context-insensitive Trusted Profile may in some cases be prevented by PathArmor. or disabled by the pointer protection component of Opaque CFI. But in general, an

IID can reveal control flow manipulation that is transparent to other EC-CFI approaches. Perhaps its main advantage is the continuous observation of program behavior without predisposition to specific patterns, making it uniquely able among CFI approaches to detect entirely unforeseen attack vectors and payload form factors, even before a fully successful exploit is carried out. An IID can even be used to assist the debugging and refinement of a CFI defense, up to its resilience against evasion tactics (Section 3.3.5), by revealing attacks and important anomalies that would otherwise go unnoticed.

The limitations of IID are also substantially distinct from its CFI counterparts. On the x86 platform, reliance on a binary translation runtime pushes overheads out of the marketable range, and creates additional hassle for installation and configuration. More importantly, profiling on any platform requires vendor support or significant investment from third-party professionals or even end users. The profiling process may be susceptible to pollution by targeted attacks, and in the case of sampling for improved performance, it may be possible for malware to detect the presence of an IID and evade it. Insufficient profiling leads to a high rate of false positives, rendering ineffective the deep introspective logging of an IID. Other forms of CFI can be deployed more securely, often with lower overhead and in many cases with lower development cost as well. These tools not only detect attacks but can also automatically prevent them before any damage is done.

It is ultimately a matter for the user to decide whether an IID merits the cost required to deploy it. A close look at the CFI competition indicates many tools could be hiding significant problems under the high level presentations that are available to the public, as suggested by the frequency of incomplete research prototypes and over-simplified evaluation scenarios. Another point to consider is that an IID can provide tangential benefits that may not be immediately evident. Research in software engineering and particularly debugging suggest that the Trusted Profile itself can be a valuable resource for understanding the real-world execution of programs that otherwise are not easily observed outside the development venue. While this presentation of IID has focused on security vulnerabilities, which are of high importance, an IID can reveal problems and unexpected factors of many kinds that are essential for efficient debugging but have proven difficult to capture in more conventional ways.

6.2 CFI in the Future Ubiquity of CET

As discussed in Section 3.1, Intel has proposed a new extension called CET to provide hardware support for securing both forward and backward control flow. It appears that this technology may not be available immediately, and that popular applications may require significant changes to become compatible with that extension. Nevertheless, it is important in the evaluation of IID to consider the potential role of IID in a future where CET does become ubiquitous on x86 devices. The presence of a hardware shadow stack would seem to take the return instructions entirely out of the attack surface. But the forward edge protection currently proposed by Intel is an EC-CFI composed of just one equivalence class, which has proven too weak for contemporary adversaries. This would imply a continued need for any defense that improves security of forward indirect branches. Section 6.1 shows that IID can be a valuable complement to any current defense in this category, and therefore should continue to be of value under the ubiquity of CET. The performance of BLACKBOX may also improve with the elimination of the software shadow stack, though other factors such as inefficient handling of the Windows IAT are likely to weigh down any hardware acceleration. In summary, although the presence of CET would make major changes in the attack surface on the x86 platform, the same fundamental security metrics would remain in effect for the protection of forward control flow edges.

6.3 Traditional Intrusion Detection

Many recent approaches to the problem of intrusion detection are successful against specific categories of vulnerabilities, such as cross-site scripting (XSS) or SQL injection (SQLi), but none of them has proven effective against remote code execution (RCE) exploits. For example, the WordPress plugin MuteScreamer [185] was once commonly used to protect WordPress sites, but it only supports a manual blacklist of request patterns, making it vulnerable to mimicry and incapable of defeating a zero-day attack. The comprehensive coverage of the trusted profile makes it possible for BLACKBOX to surpass some limitations of these otherwise successful techniques.

Input Filtering

For applications having a relatively systematic public interface, exploits can be detected with high accuracy by observing patterns in user input. Commonly deployed tools are Snort [161] and Apache ModSecurity [172], which block known attacks based on a manually maintained blacklist. But for today's complex and highly dynamic PHP applications, this approach performs poorly because (a) a single vulnerability can be compromised using distinct crafted inputs, (b) the frequency of blacklist updates would increase by several orders of magnitude (for example, `wordpress.org` currently offers over 46,000 plugins, most of which are continually in development), and (c) blacklist approaches cannot defeat zero-day attacks, yet new exploits against WordPress alone are reported almost daily.

Anomaly Detection

A variant known as *anomaly detection* relies instead on a whitelist of normal application behaviors, but the whitelist can be difficult to construct. One approach [152] requires an expert to manually define a set of policies, which also must be maintained to accommodate ongoing customization, configuration changes, and even application state changes. Several alternative techniques formulate the whitelist in simpler terms—for example, n-grams of packet bytes [179, 180], or properties of HTTP request attributes such as string length, characters sets, token sets [96, 143]—and learn the whitelist by observing a set of known-normal requests (obtained by manual analysis). Many of these approaches are subject to mimicry or evasion tactics because the whitelist is only indirectly related to program semantics. Another weakness is that the training time is typically quite long—from 8% up to 50% of the experiment's reported request set in successful techniques. Recent investigation into these approaches no longer mentions web applications, suggesting that the increasingly dynamic interaction between browser and server inhibits convergence of the whitelist. Pure machine learning approaches have mediocre results on artificial benchmarks such as the DARPA datasets [100], and have rarely been used in practice, where they typically perform much worse [162].

Introspective Anomaly Detection

An introspective n-gram approach learns a whitelist of short system call sequences that are normal for the application, identifying them by name without context or arguments [57]. This over-simplified representation of the application allows the attacker far too much flexibility, making the monitor highly susceptible to mimicry. Similar approaches based on finite state automata were evidently difficult to apply in practice, considering that no experiments were reported [148, 182].

Transition to CFI

A survey on anomaly detection techniques reports applications in a broad spectrum of domains, which includes software and network intrusions along with data-centric domains such as insurance fraud and early detection of medical conditions [25]. Several of the works mentioned here are cited in the section of the survey that focuses on intrusion detection. But although the survey includes references to works published as late as 2008, the material about intrusion detection for software applications ends in 2005. Further work focuses on a specific other than web applications or desktop software, for example intrusions that attack the network infrastructure, or techniques for better detecting such intrusions across a cluster.

A survey on intrusion detection published in 2014 includes many references to more recent research, but these works also focus on a specific context other than web applications or desktop software. Popular topics include wireless networks and medical systems [113]. A historical report by Vigna outlines the trend in application security research toward more structured approaches like CFI, showing how the difficulties with both mimicry and 0-day exploits dampened motivation in research to continue developing intrusion detection systems for user-interactive applications [176]. In his own work, Vigna published frequently on the topic of intrusion detection for user applications up until 2005, then followed this same trend toward CFI and other more systematic defense techniques.

Higher Level Intrusion Detection

Challenges in the accuracy of intrusion detection and anomaly detection have led more recent works to develop abstractions that focus detection on higher level program behaviors.

Behavior-based malware detection [106] models financially motivating *actions* taken by malicious bots. An action is defined through a layered process of abstraction, for example raising a stream of I/O system calls such as `bind` and `connect` to a stream of network events such as `async_tcp_client`, and again to program interface events such as `tcp_client` until reaching the top-layer (financially motivating) actions such as `download_file`. By focusing the lower layers of this hierarchy narrowly on essential components of the action, the model becomes robust to unexpected variations and even evasion tactics. Experiments show that the same top-layer action can be identified in a variety of bots found in different malware strains. Performance overhead exceeds 5× for any of the analyzed actions, but this can presumably improved with a more efficient platform than QEMU.

A quantitative study in accuracy [20] focuses on malware detection based on system calls, showing that while the majority of surveyed behavioral models are highly unreliable, a few approaches demonstrate extreme precision when tested against the same benchmark data sets. The models include a range of abstraction, from plain system calls to higher level actions such as “read a file” or “load a library”. Where the majority of models have less than 70% accuracy identifying a suite of malware, models based on “bags of actions with arguments” detect over 95% of the malware with fewer than 1% false positives. Here again it is the highest level of abstraction that demonstrates the best accuracy.

LEAPS [73] automatically generates models for attack detection through supervised statistical learning and leverages control flow graphs of system events to filter the models. The detector observes a stream of raw system event logs containing stack traces to learn and later distinguish logs in which an exploit occurred. In one component of the detector, a Weighted Support Vector Machine is trained on labeled logs, while in a second component the program CFG is inferred from stack traces in the logs. The detector combines this information by calculating graph distances between normal logs and attack logs, then applies the distance as a weight to the learned classifiers. Average detection rates are consistently around 90% and rarely fall below 70% for a particular strain of malware in a given program.

The Trusted Profile can be considered a similar kind of abstraction in that its elements are compared at runtime to a set of higher level operations as they occur in the running program. A key distinction from these approaches is that in IID, profiling automatically generates the Trusted Profile, whereas these approaches require some supervision or even manual modeling.

6.4 PHP Security

The challenges of static analysis on PHP (Section 5.2.3) have made it difficult to develop online defense systems for PHP applications. Instead of engaging in this uphill battle, researchers have turned their attention to the automated detection of vulnerabilities and attacks. These tools are used in practice for security debugging of applications. Some tools go so far as to generate a patch for discovered problems, but for the most part it is a manual process undertaken by the PHP developer. The majority of these tools focus on attack vectors other than remote code execution.

6.4.1 RCE Exploits vs. the PHP Platform

It is difficult to construct sophisticated low-level RCE exploits because the PHP interpreter imposes high-level structure on the execution of PHP scripts. To the best of our knowledge, the case studies in Section 5.2.2 cover the most complex takeover strategies that have presently been developed. Accordingly, research into potential attacks has mainly focused on program analysis that seeks new opportunities to employ the same old takeover strategies, for example leveraging session variables to influence program constructs that are not directly accessible via HTTP parameters. It appears that real adversaries are similarly constrained, since new attacks continue to be reported against versions of PHP application that have been in circulation for many years, yet it is rare for the deployed exploit itself to contain any novel ingenuity.

The fraction of adversarial research that focuses on RCE exploits employs static analysis to identify data flows that reach dynamic (string-based) control flow transfers. More generic tools such as advanced PHP website crawlers could also be used for this purpose, though no RCE experiments are reported or even hypothesized in these publications. Other exploits such as SQLi and XSS draw the majority of research attention, perhaps due to their greater share of reported instances in the wild.

Static Analysis

Conventional static analysis has been unable to find many important attack vectors in stateful, session-based applications, so [197] uses an SMT solver along with advanced formulation of constraints on string dataflow and program logic to find several new RCE exploits in large PHP applications. While this approach has some success, the authors admit many vulnerabilities cannot be found this way. For example, the implementation has limitations at dynamic constructs such as variable array indices, which are very common in PHP—in the version of phpMyAdmin reported in the experiment, 25% of all array accesses use a variable index.

Saner combines static and dynamic analysis to detect missing authentication checks in PHP applications, and validates each report by automatically generating a corresponding exploit. Researchers have developed static analysis for PHP to find SQL injection attacks [186] To improve coverage of a static analysis for both XSS and SQLi, [187] partitions it into 3 levels of granularity: intra-block, intra-procedural and inter-procedural. Pixy [88] employs taint tracking to find vulnerable handling of string variables.

The authors of all these techniques suggest their approaches may also work well for other kinds of exploits, including RCE—but they also admit limitations at dynamic statements such as `include` with a variable operand, which are a primary target of RCE exploits. Another limitation of these approaches is that they disregard client-side JavaScript, making it difficult for them to find all the application entry points.

Dynamic Analysis

A crawler for PHP applications named jÄk [133] discovers dynamic URL forms by installing hooks in client-side JavaScript to construct and then traverse a navigation graph of the application. Noncespaces [175] randomizes sensitive values to prevent client-side tampering. Diglossia [163] and SQLCheck [170] employ efficient taint tracking to detect SQLi.

6.5 Software Improvement Techniques

Since many exploits rely on program errors to gain control of the victim process, an alternative approach to security focuses on finding and removing errors in software. Despite major investment, it has not been possible to remove all exploitable vulnerabilities, even where the vendor controls all of the source code for the application and operating system. For example, CVE-2014-3188 allows remote code execution on a Google Chromebook via crafted JSON input. Nevertheless, both researchers and commercial engineers continue to develop a broad range of techniques that can eliminate security vulnerabilities and many other problems from software. Section 6.5.1 presents a representative selection of security-oriented debugging tools that are applied by developers as part of the software production toolchain. Section 6.5.2 follows with techniques for automatically detecting and eliminating errors at runtime.

6.5.1 Debugging

Memory debuggers AddressSanitizer [154] and MemorySanitizer [169] integrate into the LLVM compiler to instrument the application with runtime checks for correct address and buffer usage. While this approach minimizes runtime overhead, it is specific to the LLVM compiler and is also significantly less effective when applied to a subset of modules loaded by a program. Similar validation can be performed via process virtualization without these limitations using Valgrind Memcheck [156] and Dr. Memory [17].

All of these debugging approaches are limited by the coverage of the tests that are executed under the tool, since the overhead is too high for production deployment. SafeMem [141] leverages ECC-protection in hardware memory modules to detect memory leaks and memory corruption with overhead low enough for deployment (under 15%) in many consumer-facing environments. Intel has recently included bounds checking instructions in the ISA of its latest CPU architectures [33], allowing more efficient memory debugging along with runtime mitigation of memory access errors.

Other approaches expand the coverage of offline tests through a combination of symbolic execution and randomization. For example, Dr. Memory includes a fuzz testing mode [49] that repeatedly executes any selected target function with randomly generated arguments while continuously checking for memory errors. LibFuzzer [153] combines AddressSanitizer or MemorySanitizer with a fuzz testing extension of LLVM to generate similarly randomized executions. Directed Automated Randomized Testing [62] employs symbolic execution to (a) determine unvisited program paths and (b) generate inputs that guide execution down those new paths. While these approaches greatly expand test coverage, some of the generated inputs may be invalid for the test target, potentially leading to a large percentage of spurious error reports.

Without the integration of these tools into the software production cycle of major applications, it is probable that the proliferation of exposed vulnerabilities would make it impossible to perform IID profiling in adversarial conditions. There would be too much potential for malicious control flow to infect the Trusted Profile. But the highly secure application code produced under the scrutiny of these tools makes unsupervised profiling a viable option. At the same time, the ongoing reports of exploits against applications across the spectrum of runtime platforms and development toolchains suggests that a reliable monitoring approach such as Introspective Intrusion Detection remains valuable for efficiently maintaining security of deployed applications.

6.5.2 Automated Repair

Die Hard [11] provides probabilistic guarantees of memory safety by randomly padding each dynamic memory allocation requested by the application. This greatly reduces the observable problems associated with buffer overflows, dangling pointers, and other forms of heap corruption. While this does not repair problems in the source code, it has the effect of repairing the process at runtime, while incurring a slight overhead of 6% with up to 2× increased memory consumption.

Clearview [135] uses learning to patch software errors. After learning constraints on variables using Daikon [51], Clearview identifies violations of these invariants on erroneous executions and generates patches to restore the invariants. Prophet [104] first analyzes the version history of an application to learn its model of correct code, then generates patches conforming to this model. AutoFix-E [183] extracts contracts from application code to verify the semantics of its generated patches. Angelix [109] generates patches on the basis of program semantics, which is not prone to simply deleting the faulting code, and can scale to large programs such as WireShark and PHP. Experiments demonstrate that Angelix can automatically repair the OpenSSL Heartbleed vulnerability with a secure patch.

To detect missing authentication checks in PHP applications, FixMeUp [164] employs static analysis while both WebSSARI [80] and Saner [7] combine static and dynamic analysis. FixMeUp and WebSSARI automatically generate patches. Saner verifies each result by dynamically generating an exploit. ScriptGard [146] instruments both the client and server to dynamically patch faulty authentication code.

While these techniques are promising and have patched security flaws in many important applications and deployment scenarios, experimental and industrial results concur that the problem of memory errors will not be eliminated by automated repairs anytime in the near future.

6.6 Usability

Thus far it has been shown that IID can potentially improve security over existing approaches, particularly in that it can be more robust to new attacks and better able to detect evasive variations of known attacks. But the administrative effort and runtime complexity of IID still present challenges for its practical usability. This can potentially be improved by integrating work from related research areas into the IID platform. For example, the log comprehension factor of C_{adm} is fundamentally a problem of program comprehension, since the IID log contains literal control flow elements that will be difficult for most users to understand and reason about. Section 6.6.1 proposes methods for bridging the gap between low-level control flow events and the corresponding user-level features and events. Section 6.6.2 addresses the sparse coverage aspect of C_{prof} by proposing symbolic execution techniques for expanding coverage of the Trusted Profile. Since this approach has the potential to trust unsafe program paths, Section 6.6.3 discusses an alternative approach that can make a sparse Trusted Profile usable by applying machine learning to improve the suspicion ranking of anomalies.

6.6.1 Program Comprehension

While an IID does not require the source code for the monitored programs, it may be possible to leverage a source-based technique called *program comprehension* to help the administrator associate high-level program features with the low-level control flow elements that appear in the IID logs, Trusted Profile and blacklist. Dynamic approaches to program comprehension have been favored for improved accuracy [32], and are convenient for IID because it operates primarily on program traces. Much of the research in this field is oriented towards software developers and assumes the availability of a structured outline of existing features. For example, the *feature location* [48] associates source code regions with a developer-supplied set of program features. Vendors interested in improving IID support for their products can provide feature-to-source associations in a format similar to a debug symbol file (e.g., a .pdb file). Where the set of features is unknown, inference techniques such as *feature mining* [99] and *aspect mining* [13] can identify both the set of application features and the source code regions that compose them. Another technique that focuses on *execution phases* [138] may benefit the user by providing an orthogonal dimension of context. Analysis of individual executions, for example during profiling or after a high-suspicion anomaly, may benefit from *trace visualization* [31].

6.6.2 Symbolic Execution

In scenarios where it is difficult to obtain complete profiling coverage, symbolic execution is able to discover control flow paths that are reachable under any arbitrarily chosen set of execution constraints. For example, X-Force [134] traverses all paths in an x86 executable that are reachable under type-compliant inputs. By adding these paths to the Trusted Profile, false positives can be significantly reduced. But this approach may also discover some unsafe paths. In many programs, potentially malicious control flow can be reached with inputs that do respect low-level program types, and such control flow is likely to be encountered by any comprehensive symbolic execution [150]. This will not always be a problem—for example, users wishing to focus on a strict subset of potential attacks can formulate a symbolic execution that never discovers the corresponding malicious control flow, yet generates substantial coverage of the remaining control flow paths.

6.6.3 Machine Learning

In scenarios where symbolic execution might discover unwanted control flow, machine learning techniques may be able to improve the anomaly suspicion ranking enough to make a sparse Trusted Profile usable. In supervised machine learning, a classifier is automatically generated on the basis of labeled inputs, which in the case of IID would simply be safe and unsafe execution traces. Safe traces can be generated in a secure environment by running automated tests or using the application manually. For many of the attacks presented in Sections 6.1.4 and 6.4.1, the authors provide a tool to find corresponding vulnerabilities or even generate a POC against any target program. Running a successful POC will generate an unsafe trace, and in many cases even a failed POC will exhibit some unsafe behaviors. Unsafe traces occurring in the wild are equally valuable for classifier training.

Preliminary experiments with machine learning in IID showed that classifiers trained on these sources are not accurate enough to determine whether a given execution is safe, even when all traces are augmented with data flow and variable values. But similar experiments focusing on malware variants [24] and guided fuzz testing [167] suggest that such classifiers could make a significant contribution to the suspicion ranking of anomalies. Research in the computation of distance metrics covers a broad range of techniques that can be explored for reducing the classifier output to a suitable scalar value [184, 190, 98, 145]. For improved accuracy, the IID administrator can evaluate each high-suspicion anomaly by examining the control flow elements and data values that most strongly influenced its distance metric computation.

Chapter 7

Conclusions and Future Work

Introspective Intrusion Detection has the potential to detect zero-day RCE exploits that leverage entirely new and unforeseen attack vectors. It can also infer local security policies where a universal policy would either be hard to formulate, or is simply undesirable. While many important CFI defenses exist with varying degrees of automation, performance and accuracy, IID can detect control flow anomalies at consistently higher precision. The comprehensive tracing capabilities of the IID runtime can greatly assist triage efforts when an exploit does occur, and can also facilitate preventative analysis of suspicious program behaviors. An IID can be an essential tool against complex intrusions such as advanced persistent threats. Field debugging and other challenging software engineering tasks can benefit from the detailed IID logs. Prototype IID implementations for x86 COTS binaries and PHP applications demonstrate these advantages in controlled and real-world experiments, though some limitations are evident.

On the x86 platform, IID performs optimally for server applications, and is also sufficient for many interactive desktop programs. The relatively high rate of anomalies on this platform can be largely mitigated by prioritization of anomaly reports on the basis of statistical tendencies and correlation to risky system calls. Known exploits against relatively small programs are reliably detected after nominal profiling periods, and case studies show that IID can detect the most sophisticated exploits developed in research for this platform. But performance limitations and cumbersome profiling requirements indicate that further improvements could make IID substantially more usable for complex x86 programs that offer rich user interfaces and years of accumulated features.

For the relatively simple configurations of popular PHP applications, IID performs well enough for deployment in many typical usage scenarios. Authorized changes to the set of enabled application behaviors can be accurately inferred and trusted without administrative intervention. Exploits in the wild are accurately detected with a false positive ratio under .01%. But experiments with IID were limited to small application deployments that may not reveal the full range of potential challenges for IID on this platform.

A cloud-based IID Service can simplify the administrative effort of using IID, potentially handling the profiling and log management tasks transparently. Installation utilities could automatically deploy an IID and integrate it with a default IID Service configuration. An additional advantage of a central service is the opportunity to coordinate profiling across users in a circle of mutual trust, and to faciliate collaborative evaluation of anomalies.

7.1 Adoption

Despite the prevalence of security threats and the extremely high investment in software and the resources it can directly affect, security techniques are not often deployed on vulnerable systems. The security tools that are heavily used by large software organizations like Google and Microsoft typically operate behind the scenes as some part of the build cycle, such as a compiler extension or a debugging utility. This form factor is desirable because it is transparent to the released product and is not likely to cause failures at customer sites. Security tools are perceived as a risk, potentially crashing a product, or creating high overhead, or requiring attention from the end user. Even tools that can be obtained and deployed by individual users are perceived as more trouble than they are worth.

This raises an important question for the future of Introspective Intrusion Detection: since it has a significant footprint at the end-user site, requiring an entire runtime along with analysis tools, is IID viable for user adoption? No matter how much effort a vendor or security organization makes to promote the monitoring advantages of IID, if users are not willing to host it on their devices, it will sit quietly in the research closet with countless other technically qualified tools. During the experimental evaluation of the IID prototypes, we solicited several organizations whose PHP websites receive significant daily traffic, but were not able to arrange any kind of experimental collaboration. Our initial proposition was usually met with enthusiasm from members who were not directly responsible for the organization's IT infrastructure, and though the internal conversations were never shared with us, we suppose that some combination of concerns—likely including stability, manual effort and privacy—outweighed the nominal perceived benefit of the endeavor.

There is reason for optimism, however. The history of end-user security tools suggests that the slow adoption rate for new approaches is not necessarily related to usability. For example, users have suffered for decades with high overheads and pervasive false positives from anti-virus daemons, yet continue to install them and even pay license fees to keep a meager slice of the attack surface marginally protected. Tools similar to IID have been popular in the WordPress community at times. The security plugin MuteScreamer [185] integrated traditional intrusion detection into any WordPress site, though as discussed in Section 1.1, it was not possible for users or even experts to maintain effective whitelists and blacklists under the highly dynamic I/O of the WordPress client API. So although the perception of security tools is in general negative, history indicates this is not necessarily the deciding factor—users will tolerate a significant amount of inconvenience when the perceived value is well established.

Taking this into perspective, the most direct path to user adoption of IID may not begin with an arduous climb through the mountains of feature refinement, reaching toward the peaks of seamless desktop integration beyond the plateau of automated policy synchronization. A more likely route winds through the jungles of promotional networking, blazing new trails through logistical tangles and legal snares that lead eventually to volunteer experiments and live prototype deployments. In the world of software, communities rarely reach out of their comfort zone looking for new approaches, no matter how serious the problems become. This is not just a phenomenon of security, but is common for many kinds of software improvement tools. The fundamental challenge for adoption is to somehow make the new technology welcome within those communities where it can be of the most advantage.

Bibliography

[1] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity. In *12th ACM Conference on Computer and Communications Security*, CCS '05, 2005.

[2] ActiveState. Activestate poll finds enterprise developers use dynamic languages to get products to market faster and cheaper; feel less pressure about quality. `https://www.activestate.com/press-releases/activestate-poll-finds-enterprise-developers-use-dynamic-languages-get-products-marke`, accessed 2017.

[3] J. Alstott, E. Bullmore, and D. Plenz. powerlaw: A python package for analysis of heavy-tailed distributions. *PLoS ONE*, 9(1):e85777, 2014.

[4] W. Arthur, B. Mehne, R. Das, and T. Austin. Getting in control of your control flow with control-data isolation. In *CGO*, 2015.

[5] M. Athanasakis, E. Athanasopoulos, M. Polychronakis, G. Portokalidis, and S. Ioannidis. The devil is in the constants: Bypassing defenses in browser JIT engines. In *NDSS*, 2015.

[6] J. Bach. The most popular programming languages for 2017. `https://blog.appdynamics.com/engineering/the-most-popular-programming-languages-for-2017/`, 2017.

[7] D. Balzarotti, M. Cova, V. Felmetsger, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Saner: Composing static and dynamic analysis to validate sanitization in web applications. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy*, SP '08, 2008.

[8] M. Barreno, B. Nelson, A. D. Joseph, and J. D. Tygar. The security of machine learning. *Machine Learning*, 81(2):121–148, Nov 2010.

[9] beginfill.com. beginfill.com homepage. `http://beginfill.com/WebGL_Video3D`, 2014.

[10] F. Bellard. QEMU, a fast and portable dynamic translator. In *USENIX Annual Technical Conference*, 2005.

[11] E. D. Berger and B. G. Zorn. Diehard: Probabilistic memory safety for unsafe languages. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, pages 158–168, New York, NY, USA, 2006. ACM.

[12] D. Bounov, R. G. Kici, and S. Lerner. Protecting c++ dynamic dispatch through vtable interleaving. In *NDSS*, 2016.

[13] S. Breu and J. Krinke. Aspect mining using event traces. In *Proceedings of the 19th IEEE International Conference on Automated Software Engineering*, ASE '04, pages 310–315, Washington, DC, USA, 2004. IEEE Computer Society.

[14] D. Bruening. *Efficient, Transparent, and Comprehensive Runtime Code Manipulation*. PhD thesis, MIT, 2004.

[15] D. Bruening. *Efficient, Transparent and Comprehensive Runtime Code Manipulation*. PhD thesis, MIT, 2004.

[16] D. Bruening and S. Amarasinghe. Maintaining consistency and bounding capacity of software code caches. In *IEEE/ACM Symp. on Code Generation and Optimization*, 2005.

[17] D. Bruening and Q. Zhao. Practical memory checking with Dr. Memory. In *IEEE/ACM Symp. on Code Generation and Optimization*, 2011.

[18] D. Bruening, Q. Zhao, and S. Amarasinghe. Transparent dynamic instrumentation. In *ACM Conf. on Virtual Execution Environments*, 2012.

[19] N. Burow, S. A. Carr, J. Nash, P. Larsen, M. Franz, S. Brunthaler, and M. Payer. Control-flow integrity: Precision, security, and performance. *ACM Comput. Surv.*, 50(1):16:1–16:33, 2017.

[20] D. Canali, A. Lanzi, D. Balzarotti, C. Kruegel, M. Christodorescu, and E. Kirda. A quantitative study of accuracy in system call-based malware detection. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ISSTA 2012, pages 122–132, New York, NY, USA, 2012. ACM.

[21] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross. Control-flow bending: On the effectiveness of control-flow integrity. In *USENIX Security*, 2015.

[22] N. Carlini and D. Wagner. ROP is still dangerous: Breaking modern defenses. In *USENIX Security*, 2014.

[23] S. Cazzulani. Octane: The JavaScript benchmark suite for the modern web. `http://blog.chromium.org/2012/08/octane-javascript-benchmark-suite-for.html`, 2012.

[24] S. Cesare and Y. Xiang. Malware variant detection using similarity search over sets of control flow graphs. In *2011IEEE 10th International Conference on Trust, Security and Privacy in Computing and Communications*, pages 181–189, Nov 2011.

[25] V. Chandola, A. Banerjee, and V. Kumar. Anomaly detection: A survey. *ACM Comput. Surv.*, 41(3):15:1–15:58, July 2009.

[26] X. Chen, J. Andersen, Z. M. Mao, M. Bailey, and J. Nazario. Towards an understanding of anti-virtualization and anti-debugging behavior in modern malware. In *2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN)*, pages 177–186, June 2008.

[27] W. Cheng, Q. Zhao, B. Yu, and S. Hiroshige. TaintTrace: Efficient flow tracing with dynamic binary rewriting. In *IEEE Symp. on Computers and Communications*, 2006.

[28] Y. Cheng, Z. Zhou, M. Yu, X. Ding, and R. H. Deng. ROPecker: A generic and practical approach for defending against ROP attack. In *NDSS*, 2014.

[29] P. R. Cichonski, T. Millar, T. Grance, and K. Scarfone. Computer security incident handling guide. `https://www.nist.gov/publications/computer-security-incident-handling-guide`, August 2012. [Online; posted 6-August-2012].

[30] M. Conti, S. Crane, L. Davi, M. Franz, P. Larsen, M. Negro, C. Liebchen, M. Qunaibit, and A.-R. Sadeghi. Losing control: On the effectiveness of control-flow integrity under stack attacks. In *CCS*, 2015.

[31] B. Cornelissen, A. Zaidman, and A. van Deursen. A controlled experiment for program comprehension through trace visualization. *IEEE Transactions on Software Engineering*, 37(3):341–355, May 2011.

[32] B. Cornelissen, A. Zaidman, A. van Deursen, L. Moonen, and R. Koschke. A systematic survey of program comprehension through dynamic analysis. *IEEE Transactions on Software Engineering*, 35(5):684–702, Sept 2009.

[33] I. Corporation. Introduction to intel memory protection extensions. `https://software.intel.com/en-us/articles/introduction-to-intel-memory-protection-extensions`, 2013.

[34] S. J. Crane, S. Volckaert, F. Schuster, C. Liebchen, P. Larsen, L. Davi, A. Sadeghi, T. Holz, B. D. Sutter, and M. Franz. It's a TRaP: Table randomization and protection against function-reuse attacks. In *CCS*, 2015.

[35] G. F. Cretu, A. Stavrou, M. E. Locasto, S. J. Stolfo, and A. D. Keromytis. Casting out demons: Sanitizing training data for anomaly sensors. In *2008 IEEE Symposium on Security and Privacy (sp 2008)*, pages 81–95, May 2008.

[36] J. Criswell, N. Dautenhahn, and V. Adve. Kcofi: Complete control-flow integrity for commodity operating system kernels. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, SP '14, pages 292–307, Washington, DC, USA, 2014. IEEE Computer Society.

[37] J. Criswell, A. Lenharth, D. Dhurjati, and V. Adve. Secure virtual architecture: A safe execution environment for commodity operating systems. *SIGOPS Oper. Syst. Rev.*, 41(6):351–366, Oct. 2007.

[38] CXSecurity. WordPress revslider arbitrary file upload, download & cross site scripting. https://cxsecurity.com/issue/WLB-2015060136, 2015.

[39] CXSecurity. Wordpress formcraft plugin file upload vulnerability. https://cxsecurity.com/issue/WLB-2016020136, 2016.

[40] A. Dahmani. How many wordpress plugins are there? https://www.adame.ma/how-many-wordpress-plugins, 2016.

[41] J. Dahse, N. Krein, and T. Holz. Code reuse attacks in php: Automated pop chain generation. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, 2014.

[42] L. Davi, R. Dmitrienko, M. Egele, T. Fischer, T. Holz, R. Hund, S. Nürnberger, and A. reza Sadeghi. MoCFI: A framework to mitigate control-flow attacks on smartphones. In *Proceedings of the Network and Distributed System Security Symposium*, 2012.

[43] L. Davi, A.-R. Sadeghi, D. Lehmann, and F. Monrose. Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection. In *USENIX Security*, 2014.

[44] J. C. Dehnert, B. K. Grant, J. P. Banning, R. Johnson, T. Kistler, A. Klaiber, and J. Mattson. The Transmeta Code Morphing Software$^{\text{TM}}$: Using speculation, recovery, and adaptive retranslation to address real-life challenges. In *IEEE/ACM Symp. on Code Generation and Optimization*, 2003.

[45] D. Dhurjati and V. Adve. Efficiently detecting all dangling pointer uses in production servers. In *IEEE/IFIP Conf. on Dependable Systems and Networks*, 2006.

[46] D. Dhurjati, S. Kowshik, and V. Adve. Safecode: Enforcing alias analysis for weakly typed languages. *SIGPLAN Not.*, 41(6):144–157, June 2006.

[47] P. Dinges and G. Agha. Targeted test input generation using symbolic-concrete backward execution. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, pages 31–36, New York, NY, USA, 2014. ACM.

[48] B. Dit, M. Revelle, M. Gethers, and D. Poshyvanyk. Feature location in source code: a taxonomy and survey. *Journal of Software: Evolution and Process*, 25(1):53–95, 2013.

[49] drmemory.org. Dr. Memory fuzz testing mode. http://drmemory.org/docs/page_fuzzer.html, 2015.

[50] dynamorio.org. Register usage coordinator. http://dynamorio.org/docs/group__drreg.html, accessed 2017.

[51] M. D. Ernst, W. G. Griswold, Y. Kataoka, and D. Notkin. Dynamically discovering pointer-based program invariants. Technical Report UW-CSE-99-11-02, University of Washington Department of Computer Science and Engineering, Seattle, WA, November 16, 1999. Revised March 17, 2000.

[52] S. Esser. Utilizing code reuse or return oriented programming in PHP applications. In *Black Hat, USA*, 2010.

[53] I. Evans, S. Fingeret, J. Gonzalez, U. Otgonbaatar, T. Tang, H. Shrobe, S. Sidiroglou-Douskos, M. Rinard, and H. Okhravi. Missing the point(er): On the effectiveness of code pointer integrity. In *S&P*, 2015.

[54] I. Evans, F. Long, U. Otgonbaatar, H. Shrobe, M. Rinard, H. Okhravi, and S. Sidiroglou-Douskos. Control jujutsu: On the weaknesses of fine-grained control flow integrity. In *CCS*, 2015.

[55] V. Feldman. 540 million active plugins makes wordpress a billion dollar market. `https://freemius.com/blog/540-million-active-plugins-makes-wordpress-a-billion-dollar-market`, 2015.

[56] F. Fischer, K. Böttinger, H. Xiao, C. Stransky, Y. Acar, M. Backes, and S. Fahl. Stack overflow considered harmful? the impact of copy&paste on android application security. In *Symposium on Security and Privacy*, Oakland'17. IEEE, 2017.

[57] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff. A sense of self for unix processes. In *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, SP '96, 1996.

[58] T. Garfinkel and M. Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *In Proc. Network and Distributed Systems Security Symposium*, pages 191–206, 2003.

[59] R. Gawlik and T. Holz. Towards automated integrity protection of c++ virtual function tables in binary programs. In *Proceedings of the 30th Annual Computer Security Applications Conference*, ACSAC '14, pages 396–405, New York, NY, USA, 2014. ACM.

[60] X. Ge, W. Cui, and T. Jaeger. Griffin: Guarding control flows using intel processor trace. *SIGARCH Comput. Archit. News*, 45(1):585–598, Apr. 2017.

[61] X. Ge, N. Talele, M. Payer, and T. Jaeger. Fine-grained control-flow integrity for kernel software. In *2016 IEEE European Symposium on Security and Privacy (EuroS P)*, pages 179–194, March 2016.

[62] P. Godefroid, N. Klarlund, and K. Sen. Dart: Directed automated random testing. *SIGPLAN Not.*, 40(6):213–223, June 2005.

[63] E. Göktas, E. Athanasopoulos, H. Bos, and G. Portokalidis. Out of control: Overcoming control-flow integrity. In *S&P*, 2014.

[64] E. Göktas, E. Athanasopoulos, M. Polychroniakis, H. Bos, and G. Portokalidis. Size does matter - why using gadget chain length to prevent code-reuse attacks is hard. In *USENIX Security*, 2014.

[65] Goo Labs. Goo Labs homepage. http://labs.gooengine.com/videosphere, 2014.

[66] Google, Inc. V8 JavaScript engine. https://code.google.com/p/v8/, 2014.

[67] Google, Inc. Google trends: Wordpress plugins. https://trends.google.com/trends/explore/TIMESERIES?date=all&q=wordpress%20plugins&hl=en-US&sni=1, 2017.

[68] Google, Inc. Google trends: Wordpress security. https://trends.google.com/trends/explore/TIMESERIES?date=all&q=wordpress%20security&hl=en-US&sni=1, 2017.

[69] Google, Inc. Google trends: Wordpress themes. https://trends.google.com/trends/explore/TIMESERIES?date=all&q=wordpress%20plugins&hl=en-US&sni=1, 2017.

[70] W. S. Gosset. The probable error of a mean. *Biometrika*, 6(1), 1908.

[71] J. L. Greathouse, C. LeBlanc, T. Austin, and V. Bertacco. Highly scalable distributed dataflow analysis. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '11, pages 277–288, Washington, DC, USA, 2011. IEEE Computer Society.

[72] J. L. Greathouse, I. Wagner, D. A. Ramos, G. Bhatnagar, T. Austin, V. Bertacco, and S. Pettie. Testudo: Heavyweight security analysis via statistical sampling. In *Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 41, pages 117–128, Washington, DC, USA, 2008. IEEE Computer Society.

[73] Z. Gu, K. Pei, Q. Wang, L. Si, X. Zhang, and D. Xu. Leaps: Detecting camouflaged attacks with statistical learning guided by program analysis. In *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 57–68, June 2015.

[74] B. Hawkins, B. Demsky, D. Bruening, and Q. Zhao. Optimizing binary translation for dynamically generated code. In *CGO*, 2015.

[75] J. L. Henning. SPEC CPU2006 benchmark descriptions. *SIGARCH Computer Architecture News*, 2006.

[76] C. Herley and P. C. van Oorschot. Sok: Science, security and the elusive goal of security as a scientific pursuit. In *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*, pages 99–120, 2017.

[77] A. Homescu, S. Brunthaler, P. Larsen, and M. Franz. Librando: transparent code randomization for just-in-time compilers. In *ACM Conf. on Computer & Communications Security*, 2013.

[78] A. Homescu, S. Neisius, P. Larsen, S. Brunthaler, and M. Franz. Profile-guided automated software diversity. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, CGO '13, pages 1–11, Washington, DC, USA, 2013. IEEE Computer Society.

[79] P. Hu, H. Li, H. Fu, D. Cansever, and P. Mohapatra. Dynamic defense strategy against advanced persistent threat with insiders. In *INFOCOM*, pages 747–755. IEEE, 2015.

[80] Y.-W. Huang, F. Yu, C. Hang, C.-H. Tsai, D.-T. Lee, and S.-Y. Kuo. Securing web application code by static analysis and runtime protection. In *Proceedings of the 13th International Conference on World Wide Web*, WWW '04, 2004.

[81] E. M. Hutchins, M. J. Cloppert, and R. M. Amin. Intelligence-driven computer network defense informed by analysis of adversary campaigns and intrusion kill chains. *Leading Issues in Information Warfare & Security Research*, April 2014. [Online; posted 3-April-2014].

[82] Intel, Corp. Control-flow enforcement technology preview. `https://software.intel.com/sites/default/files/managed/4d/2a/control-flow-enforcement-technology-preview.pdf`, 2017.

[83] Intel Corp. Media alert: New 8th gen intel core processor family to debut aug. 21. `https://newsroom.intel.com/news-releases/media-alert-introducing-new-8th-gen-intel-core-processor-family/`, 2017.

[84] Intel Corp. Products formerly kaby lake. `https://ark.intel.com/products/codename/82879/Kaby-Lake`, accessed 2017.

[85] interrupt3@mail.ru. Tomahawk multipass morpher engine. Transitory malware development forum (URL removed), 2010–2012.

[86] interrupt3@mail.ru. Krypton (malware development template). Transitory malware development forum (URL removed), [obtained in 2013].

[87] D. Jang, Z. Tatlock, and S. Lerner. SafeDispatch: Securing C++ virtual calls from memory corruption attacks. In *Proceedings of the 2014 Network and Distributed System Security Symposium*, NDSS '14, 2014.

[88] N. Jovanovic, C. Kruegel, and E. Kirda. Pixy: A static analysis tool for detecting web application vulnerabilities (short paper). In *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, SP '06, 2006.

[89] B. Kasikci, B. Schubert, C. Pereira, G. Pokam, and G. Candea. Failure sketching: A technique for automated root cause diagnosis of in-production failures. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 344–360, New York, NY, USA, 2015. ACM.

[90] D. Keppel. A portable interface for on-the-fly instruction space modification. In *ACM Conf. on Architectural Support for Programming Languages and Operating Systems*, 1991.

[91] D. Keppel. How to detect self-modifying code during instruction-set simulation. In *IEEE/ACM Workshop on Architectural and Microarchitectural Support for Binary Translation*, 2009.

[92] C. Kerschbaumer, E. Hennigan, P. Larsen, S. Brunthaler, and M. Franz. Crowdflow: Efficient information flow security. In *Proceedings of the 16th International Conference on Information Security - Volume 7807*, ISC 2013, pages 321–337, New York, NY, USA, 2015. Springer-Verlag New York, Inc.

[93] F. M. Kifetew. A search-based framework for failure reproduction. In *Proceedings of the 4th International Conference on Search Based Software Engineering*, SSBSE'12, pages 279–284, Berlin, Heidelberg, 2012. Springer-Verlag.

[94] V. Kiriansky, D. Bruening, and S. Amarasinghe. Secure execution via program shepherding. In *Proceedings of the 11th USENIX Security Symposium*, USENIX Security '02, 2002.

[95] A. J. Ko and B. A. Myers. Designing the whyline: A debugging interface for asking questions about program behavior. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '04, pages 151–158, New York, NY, USA, 2004. ACM.

[96] C. Kruegel and G. Vigna. Anomaly detection of web-based attacks. In *Proceedings of the 10th ACM Conference on Computer and Communications Security*, CCS '03, 2003.

[97] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song. Code-pointer integrity. In *OSDI*, 2014.

[98] J. T. Kwok and I. W. Tsang. Learning with idealized kernels. In *Proceedings of the Twentieth International Conference on International Conference on Machine Learning*, ICML'03, pages 400–407. AAAI Press, 2003.

[99] C. Kästner, A. Dreiling, and K. Ostermann. Variability mining: Consistent semi-automatic detection of product-line features. *IEEE Transactions on Software Engineering*, 40(1):67–82, Jan 2014.

[100] M. L. Laboratories. DARPA intrusion detection data sets. `https://www.ll.mit.edu/ideval/data/`.

[101] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz. Sok: Automated software diversity. In *S&P*, 2014.

[102] A. Lazarevic and V. Kumar. Feature bagging for outlier detection. In *Proceedings of the Eleventh ACM SIGKDD International Conference on Knowledge Discovery in Data Mining*, KDD '05, pages 157–166, New York, NY, USA, 2005. ACM.

[103] J. Li, C. Wu, and W.-C. Hsu. Dynamic register promotion of stack variables. In *IEEE/ACM Symp. on Code Generation and Optimization*, 2011.

[104] F. Long and M. Rinard. Automatic patch generation by learning correct code. *SIGPLAN Not.*, 51(1):298–312, Jan. 2016.

[105] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *ACM Conf. on Programming Language Design and Implementation*, 2005.

[106] L. Martignoni, E. Stinson, M. Fredrikson, S. Jha, and J. C. Mitchell. A layered architecture for detecting malicious behaviors. In *Proceedings of the 11th International Symposium on Recent Advances in Intrusion Detection*, RAID '08, pages 78–97, Berlin, Heidelberg, 2008. Springer-Verlag.

[107] A. J. Mashtizadeh, A. Bittau, D. Boneh, and D. Mazières. Ccfi: Cryptographically enforced control flow integrity. In *CCS*, 2015.

[108] P. Maydell. Inquiry, QEMU developer's email list. `http://lists.gnu.org/archive/html/qemu-devel/2014-08/msg05142.html`, 2014.

[109] S. Mechtaev, J. Yi, and A. Roychoudhury. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 691–701, May 2016.

[110] Microsoft. A detailed description of the data execution prevention (DEP) feature in Windows XP Service Pack 2, Windows XP Tablet PC Edition 2005, and Windows Server 2003. `support.microsoft.com/kb/875352`.

[111] M. Miller. Mitigating arbitrary native code execution in microsoft edge. `https://blogs.windows.com/msedgedev/2017/02/23/mitigating-arbitrary-native-code-execution/`, 2017.

[112] N. Miramirkhani, M. P. Appini, N. Nikiforakis, and M. Polychronakis. Spotless sandboxes: Evading malware analysis systems using wear-and-tear artifacts. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 1009–1024, May 2017.

[113] R. Mitchell and I.-R. Chen. A survey of intrusion detection techniques for cyber-physical systems. *ACM Comput. Surv.*, 46(4):55:1–55:29, Mar. 2014.

[114] V. Mohan, P. Larsen, S. Brunthaler, K. W. Hamlen, and M. Franz. Opaque control-flow integrity. In *NDSS*, 2015.

[115] A. Mokarian, A. Faraahi, and A. G. Delavar. False positives reduction techniques in intrusion detection systems-a review. *International Journal of Computer Science and Network Security (IJCSNS)*, 13(10):128, 2013.

[116] N. Mor-Sarid. Inquiry, Pinheads email list. `https://groups.yahoo.com/neo/groups/pinheads/conversations/messages/10959`, 2014.

[117] Mozilla. IonMonkey. `https://wiki.mozilla.org/IonMonkey`, 2014.

[118] Mozilla. Kraken JavaScript benchmark (version 1.1). `http://krakenbenchmark.mozilla.org/`, 2014.

[119] S. Murtaza, A. Hamou-Lhadj, and M. Couture. Reducing false positive rate in anomaly detection through generalization of system calls. `http://pubs.drdc-rddc.gc.ca/BASIS/pcandid/www/frepub/DDW?W%3DSYSNUM=536524&r=0`, October 2011. [Online; posted 1-October-2011].

[120] N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *ACM Conf. on Programming Language Design and Implementation*, 2007.

[121] F. Ng. A story of cyber attack and incident response. `https://www.linkedin.com/pulse/story-cyber-attack-incident-response-freeman-ng`, 2015.

[122] B. Niu and G. Tan. Monitor integrity protection with space efficiency and separate compilation. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer &#38; Communications Security*, CCS '13, pages 199–210, New York, NY, USA, 2013. ACM.

[123] B. Niu and G. Tan. Modular control-flow integrity. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 577–587, New York, NY, USA, 2014. ACM.

[124] B. Niu and G. Tan. Rockjit: Securing just-in-time compilation using modular control-flow integrity. In *ACM Conf. on Computer and Communications Security*, 2014.

[125] B. Niu and G. Tan. Per-input control-flow integrity. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 914–926, New York, NY, USA, 2015. ACM.

[126] P. Ohmann, A. Brooks, L. D&#039;Antoni, and B. Liblit. Control-flow recovery from partial failure reports. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, pages 390–405, New York, NY, USA, 2017. ACM.

[127] P. Ohmann and B. Liblit. Lightweight control-flow instrumentation and postmortem analysis in support of debugging. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 378–388, Nov 2013.

[128] V. Pappas, M. Polychronakis, and A. D. Keromytis. Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, SP '12, 2012.

[129] V. Pappas, M. Polychronakis, and A. D. Keromytis. Transparent ROP exploit mitigation using indirect branch tracing. In *SEC*, 2013.

[130] G. A. Pascoe. Elements of object-oriented programming. *BYTE*, 11(8):139–144, Aug. 1986.

[131] M. Payer, A. Barresi, and T. R. Gross. Fine-grained control-flow integrity through binary hardening. In *Proceedings of the 12th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment - Volume 9148*, DIMVA 2015, pages 144–164, New York, NY, USA, 2015. Springer-Verlag New York, Inc.

[132] M. Payer and T. R. Gross. Fine-grained user-space security through virtualization. *SIGPLAN Not.*, 46(7):157–168, Mar. 2011.

[133] G. Pellegrino, C. Tschürtz, E. Bodden, and C. Rossow. *jÄk: Using Dynamic Analysis to Crawl and Test Modern Web Applications*, pages 295–316. Springer International Publishing, Cham, 2015.

[134] F. Peng, Z. Deng, X. Zhang, D. Xu, Z. Lin, and Z. Su. X-Force: Force-executing binary programs for security applications. In *USENIX Security*, 2014.

[135] J. H. Perkins, S. Kim, S. Larsen, S. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, S. Sidiroglou, G. Sullivan, W.-F. Wong, Y. Zibin, M. D. Ernst, and M. Rinard. Automatically patching errors in deployed software. In *SOSP*, 2009.

[136] J. Pewny and T. Holz. Control-flow restrictor: Compiler-based cfi for ios. In *Proceedings of the 29th Annual Computer Security Applications Conference*, ACSAC '13, pages 309–318, New York, NY, USA, 2013. ACM.

[137] T. Pietraszek and A. Tanner. Data mining and machine learning-towards reducing false positives in intrusion detection. *Inf. Secur. Tech. Rep.*, 10(3):169–183, Jan. 2005.

[138] H. Pirzadeh, A. Agarwal, and A. Hamou-Lhadj. An approach for detecting execution phases of a system for the purpose of program comprehension. In *2010 Eighth ACIS International Conference on Software Engineering Research, Management and Applications*, pages 207–214, May 2010.

[139] B. P. Pokkunuri. Object oriented programming. *SIGPLAN Not.*, 24(11):96–101, Nov. 1989.

[140] A. Prakash, X. Hu, and H. Yin. vfGuard: Strict protection for virtual function calls in cots c++ binaries. In *NDSS*, 2015.

[141] F. Qin, S. Lu, and Y. Zhou. Safemem: exploiting ecc-memory for detecting memory leaks and memory corruption during production runs. In *11th International Symposium on High-Performance Computer Architecture*, pages 291–302, Feb 2005.

[142] F. Richter. Landline phones are a dying breed. `https://www.statista.com/chart/2072/landline-phones-in-the-united-states/`, 2017.

[143] W. Robertson, G. Vigna, C. Kruegel, and R. A. Kemmerer. Using generalization and characterization techniques in the anomaly-based detection of web attacks. In *In Proceedings of the 13 th Symposium on Network and Distributed System Security (NDSS)*, 2006.

[144] N. Rosenblum, X. Zhu, B. Miller, and K. Hunt. Machine learning-assisted binary code analysis. In *NIPS Workshop on Machine Learning in Adversarial Environments for Computer Security, Whistler, British Columbia, Canada, December*, 2007.

[145] L. K. Saul and S. T. Roweis. Think globally, fit locally: Unsupervised learning of low dimensional manifolds. *J. Mach. Learn. Res.*, 4:119–155, Dec. 2003.

[146] P. Saxena, D. Molnar, and B. Livshits. SCRIPTGARD: Automatic context-sensitive sanitization for large-scale legacy web applications. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, 2011.

[147] K. Scarfone and P. M. Mell. Guide to intrusion detection and prevention systems (idps). `https://www.nist.gov/publications/guide-intrusion-detection-and-prevention-systems-idps`, February 2007. [Online; posted 20-February-2007].

[148] F. B. Schneider. Enforceable security policies. volume 3, pages 30–50, Feb. 2000.

[149] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A.-R. Sadeghi, and T. Holz. Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in C++ applications. In *S&P*, 2015.

[150] E. J. Schwartz, T. Avgerinos, and D. Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, pages 317–331, Washington, DC, USA, 2010. IEEE Computer Society.

[151] M. J. Schwartz. Social engineering leads apt attack vectors. `https://www.darkreading.com/vulnerabilities-and-threats/social-engineering-leads-apt-attack-vectors/d/d-id/1100142`, September 2011. [Online; posted 14-September-2011].

[152] D. Scott and R. Sharp. Abstracting application-level web security. In *Proceedings of the 11th International Conference on World Wide Web*, WWW '02, 2002.

[153] K. Serebryany. Continuous fuzzing with libfuzzer and addresssanitizer. In *2016 IEEE Cybersecurity Development (SecDev)*, pages 157–157, Nov 2016.

[154] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov. Addresssanitizer: A fast address sanity checker. In *Presented as part of the 2012 USENIX Annual Technical Conference (USENIX ATC 12)*, pages 309–318, Boston, MA, 2012. USENIX.

[155] K. Serebryany and T. Iskhodzhanov. Threadsanitizer: Data race detection in practice. In *ACM Workshop on Binary Instrumentation and Applications*, 2009.

[156] J. Seward and N. Nethercote. Using Valgrind to detect undefined value errors with bit-precision. In *USENIX Annual Technical Conference*, 2005.

[157] A. Shah. 10 evergreen programming languages to learn in 2015. `https://blog.fusioninformatics.com/new-technology-trends/10-evergreen-programming-languages-learn-2015/`, 2015.

[158] R. C. Sharble and S. S. Cohen. The object-oriented brewery: A comparison of two object-oriented development methods. *SIGSOFT Softw. Eng. Notes*, 18(2):60–73, Apr. 1993.

[159] sibaway7@yahoo.com. Krypton icon morphing plugin. Transitory malware development forum (URL removed), [obtained in 2013].

[160] M. Siegler. Chrome appears to have hit 10,000 extensions, inching closer to firefox. `https://techcrunch.com/2010/12/10/chrome-extension-numbers/`, 2010.

[161] snort.org. Snort - network intrusion detection & prevention system. `https://www.snort.org/`.

[162] R. Sommer and V. Paxson. Outside the closed world: On using machine learning for network intrusion detection. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, 2010.

[163] S. Son, K. S. McKinley, and V. Shmatikov. Diglossia: Detecting code injection attacks with precision and efficiency. In *Proceedings of the 2013 ACM Conference on Computer and Communications Security*, 2013.

[164] S. Son, K. S. McKinley, and V. Shmatikov. FixMeUp: Repairing access-control bugs in web applications. In *Proceedings of the 2013 Network and Distributed System Security Symposium*, 2013.

[165] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena. BitBlaze: A new approach to computer security via binary analysis. In *IEEE International Conf. on Information Systems Security*, 2008.

[166] E. O. Soremekun. Debugging with probabilistic event structures. In *Proceedings of the 39th International Conference on Software Engineering Companion*, ICSE-C '17, pages 437–440, Piscataway, NJ, USA, 2017. IEEE Press.

[167] S. Sparks, S. Embleton, R. Cunningham, and C. Zou. Automated vulnerability analysis: Leveraging control flow for evolutionary input crafting. In *Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007)*, pages 477–486, Dec 2007.

[168] Standard Performance Evaluation Corporation. SPEC CPU2006 benchmark suite. `http://www.spec.org/osg/cpu2006/`, 2006.

[169] E. Stepanov and K. Serebryany. Memorysanitizer: Fast detector of uninitialized memory use in c++. In *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '15, pages 46–55, Washington, DC, USA, 2015. IEEE Computer Society.

[170] Z. Su and G. Wassermann. The essence of command injection attacks in web applications. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '06, 2006.

[171] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, Úlfar Erlingsson, L. Lozano, and G. Pike. Enforcing forward-edge control-flow integrity in GCC & LLVM. In *USENIX Security*, 2014.

[172] Trustwave SpiderLabs. ModSecurity: Open source web application firewall. https://www.modsecurity.org/.

[173] TSan. ThreadSanitizer wiki. https://code.google.com/p/data-race-test/source/browse/trunk/dynamic_annotations/dynamic_annotations.h, 2014.

[174] V. van der Veen, D. Andriesse, E. Göktaş, B. Gras, L. Sambuc, A. Slowinska, H. Bos, and C. Giuffrida. Practical context-sensitive CFI. In *CCS*, 2015.

[175] M. Van Gundy and H. Chen. Noncespaces: Using randomization to defeat cross-site scripting attacks. volume 31, pages 612–628. Elsevier Advanced Technology Publications, June 2012.

[176] G. Vigna. Network intrusion detection: Dead or alive? In *Proceedings of the 26th Annual Computer Security Applications Conference*, ACSAC '10, pages 117–126, New York, NY, USA, 2010. ACM.

[177] C. Viviani. WordPress Download Manager 2.7.4 - remote code execution vulnerability. https://www.exploit-db.com/exploits/35533/, 2015.

[178] N. Walkinshaw, S. Afshan, and P. McMinn. Using compression algorithms to support the comprehension of program traces. In *Proceedings of the Eighth International Workshop on Dynamic Analysis*, WODA '10, pages 8–13, New York, NY, USA, 2010. ACM.

[179] K. Wang, G. Cretu, and S. J. Stolfo. Anomalous payload-based worm detection and signature generation. In *Proceedings of the 8th International Conference on Recent Advances in Intrusion Detection*, RAID'05.

[180] K. Wang, J. J. Parekh, and S. J. Stolfo. Anagram: A content anomaly detector resistant to mimicry attack. In *Proceedings of the 9th International Conference on Recent Advances in Intrusion Detection*, RAID'06, 2006.

[181] Z. Wang and X. Jiang. Hypersafe: A lightweight approach to provide lifetime hypervisor control-flow integrity. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, pages 380–395, Washington, DC, USA, 2010. IEEE Computer Society.

[182] G. Wassermann and Z. Su. Sound and precise analysis of web applications for injection vulnerabilities. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07.

[183] Y. Wei, Y. Pei, C. A. Furia, L. S. Silva, S. Buchholz, B. Meyer, and A. Zeller. Automated fixing of programs with contracts. In *Proceedings of the 19th International Symposium on Software Testing and Analysis*, ISSTA '10, pages 61–72, New York, NY, USA, 2010. ACM.

[184] K. Q. Weinberger and L. K. Saul. Distance metric learning for large margin nearest neighbor classification. *J. Mach. Learn. Res.*, 10:207–244, June 2009.

[185] WordPress. Mute screamer—wordpress plugins. https://wordpress.org/plugins/mute-screamer/.

[186] Y. Xie and A. Aiken. Static detection of security vulnerabilities in scripting languages. In *Proceedings of the 15th Conference on USENIX Security Symposium - Volume 15*, 2006.

[187] Y. Xie and A. Aiken. Static detection of security vulnerabilities in scripting languages. In *Proceedings of the 15th Conference on USENIX Security Symposium - Volume 15*, USENIX-SS'06, 2006.

[188] J. Xu, D. Mu, P. Chen, X. Xing, P. Wang, and P. Liu. Credal: Towards locating a memory corruption vulnerability with your core dump. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 529–540. ACM, 2016.

[189] J. Xu, D. Mu, X. Xing, P. Liu, P. Chen, and B. Mao. Postmortem program analysis with hardware-enhanced post-crash artifacts. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 17–32, Vancouver, BC, 2017. USENIX Association.

[190] L. Yang, R. Jin, R. Sukthankar, and Y. Liu. An efficient algorithm for local distance metric learning. In *Proceedings of the 21st National Conference on Artificial Intelligence - Volume 1*, AAAI'06, pages 543–548. AAAI Press, 2006.

[191] D. Yuan, S. Park, P. Huang, Y. Liu, M. M.-J. Lee, X. Tang, Y. Zhou, and S. Savage. Be conservative: Enhancing failure diagnosis with proactive logging. In *OSDI*, volume 12, pages 293–306, 2012.

[192] P. Yuan, Q. Zeng, and X. Ding. Hardware-assisted fine-grained code-reuse attack detection. In *Proceedings of the 18th International Symposium on Research in Attacks, Intrusions, and Defenses - Volume 9404*, RAID 2015, pages 66–85, New York, NY, USA, 2015. Springer-Verlag New York, Inc.

[193] C. Zamfir and G. Candea. Execution synthesis: A technique for automated software debugging. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, pages 321–334, New York, NY, USA, 2010. ACM.

[194] C. Zhang, C. Song, K. Z. Chen, Z. Chen, and D. X. Song. VTint: Protecting virtual function tables' integrity. In *NDSS*, 2015.

[195] C. Zhang, T. Wei, Z. Chen, L. Duan, S. McCamant, L. Szekeres, D. Song, and W. Zou. Practical control flow integrity & randomization for binary executables. In *Proceedings of IEEE Symposium on Security and Privacy*, 2013.

[196] M. Zhang and R. Sekar. Control flow integrity for COTS binaries. In *USENIX Security Symposium*, 2013.

[197] Y. Zheng and X. Zhang. Path sensitive static analysis of web applications for remote code execution vulnerability detection. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, 2013.

[198] C. C. Zou and R. Cunningham. Honeypot-aware advanced botnet construction and maintenance. In *International Conference on Dependable Systems and Networks (DSN'06)*, pages 199–208, June 2006.

Appendix A

Index of Implementation Concerns

List of Advantages

List of Challenges

List of Platform Snares

Derivation	Motivation and Rationale
G_{kx}	**kx** = detect **k**nown e**x**ploits
	An IID should detect any known RCE exploit against real-world applications without relying on predefined knowledge of its takeover strategy.
A_{wpm}	Whole process monitoring without anticipating specific kinds of attacks.
C_{prof}	Profiling can be tuned using known exploit POCs.
G_{rx}	**rx** = detect e**x**ploits developed in **r**esearch
	An IID should detect sophisticated RCE exploits developed in research.
A_{wpm}	Only pure reuse attacks should be transparent to IID.
C_{prof}	Verify the design of the Trusted Profile against research exploits.
G_{wx}	**wx** = detect e**x**ploits in the **w**ild
	An IID should detect exploits occurring in the wild.
A_{wpm}	IID should remain effective under real-world variations in execution.
C_{prof}	Verify the Trusted Profile against exploits occurring in the wild.
G_{dnd}	**dnd** = **d**o **n**ot **d**isturb
	Given a reasonable effort to generate a valid Trusted Profile, spurious anomalies should rarely or never be reported.
A_{log}	IID logs should reliably highlight suspicious program behavior.
C_{prof}	The Trusted Profile should model execution on the target platform in a way that facilitates effective profiling and monitoring.
G_{mod}	**mod** = **mod**ify the Trusted Profile or blacklist
	It should be easy to modify the IID security policy, for example by pasting a log entry into either the Trusted Profile or the blacklist.
A_{log}	Log entries can be precise enough for use as control flow specifications.
A_{acc}	The Trusted Profile and blacklist should be accessible to the user.
A_{loc}	The IID security policies are local to the deployment.
C_{prof}	Minimize the effort required to maintain an effective Trusted Profile.

Derivation	Motivation and Rationale
G_{dbg}	**dbg** = provide effective **deb**u**g**ging assistance
	The IID runtime should be configurable for expanded reporting to support the debugging usage scenario.
A_{wpm}	An IID report can include any aspect of the execution, including information that is only visible before or after an anomaly occurs.
A_{log}	Introspection facilitates precise control flow logging.
G_{blk}	**blk** = effective **bl**ac**k**list design and implementation
	All aspects of an IID should be carefully considered in the design of blacklist features and the implementation of blacklist operations.
A_{acc}	Many use cases are made possible by access to IID policy.
A_{loc}	The potential burden of local policies should also be considered.
G_{lite}	**lite** = runtime overhead should be **lite**
	IID runtime overhead should not drastically change the user experience of the monitored program or introduce esoteric hardware requirements.
C_{perf}	Hardware-based execution platforms require a separate, dedicated runtime that is prone to overhead.
C_{impl}	Usable performance should not rely on rocket science.
G_{dep}	**dep** = user **dep**loyable
	It should be possible for non-expert users to securely deploy the IID without having direct assistance from an expert.
C_{adm}	An IID will be installed by typical developers and administrators.
G_{dev}	**dev** = reasonable **dev**elopment cost
	A successful IID should maintain a cost/benefit ratio that is practical, even for initial deployments of the technology.
C_{impl}	The development cost of an IID can become prohibitive, potentially leading to compromised goals or even project failure.

Table A.1: Requirements diagram of the IID goals, for reference and cross-validation during design and implementation. It is always important to account for the advantages and challenges of IID.

Appendix B

Implementing IID for a New Platform

As presented in Chapters 3 and 5, each runtime platform is likely to bring a unique set of complications to the design and implementation of a new IID. Appendix A provides an index into the advantages and challenges of IID in general, the goals that have been derived from those challenges, and the snares occurring on the x86 and PHP platforms. The first step toward implementing a new IID is to identify the snares for the new platform, which will simply be those platform characteristics that are not immediately compatible with the IID goals.

The next step is to formulate a Trusted Profile that captures the security sensitive elements of application behavior on the platform. Given that today's computer science cannot offer a concrete definition of what it means for an application to be secure (Chapter 6), the set of security sensitive elements for a platform can only be derived from the exploits known to occur against its applications. This makes formulation of the Trusted Profile a fairly subjective process. For this reason, a Trusted Profile design should be verified by testing, first by manually tracing the IID response to abstractions of known exploits, and eventually by benchmarking detection of real exploit instances.

If the two IID implementations presented here are indeed representative of runtime platforms in general, then any remaining work should essentially be a matter of conventional software engineering.

It will be necessary to maintain the complete set of IID goals while addressing individual factors such as performance, application compatibility, layers of abstraction and profiling coverage. But these goals can simply be accounted for as additional constraints in the IID specification. From this perspective, the single fundamental challenge in implementing an IID for a particular platform is just to decide what constitutes a safe execution of a program running in that environment.