UNIVERSITY OF CALIFORNIA,
IRVINE

DCFI: Control Flow Integrity for Modern Windows Applications

THESIS

submitted in partial satisfaction of the requirements
for the degree of

MASTER OF SCIENCE

in Electrical and Computer Engineering

by

Byron Hawkins

Thesis Committee:
Professor Brian Demsky, Chair
Professor Athina Markopoulou
Professor Harry Xu

2014

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGMENTS

I would like to thank...

1. Professor Brian Demsky for initiating this project and establishing its technical direction throughout the course of its development.
2. Professors Harry Xu and Michael Taylor for their helpful critique and suggestions regarding the presentation of these research ideas.
3. Peizhao Ou, for developing the original CFG merging and analysis program (and for analyzing all kinds of weird malware graphs).
4. DynamoRIO developers Derek Bruening, Qin Zhao and Reid Kleckner for their help with my most difficult technical questions (and for developing a great virtual machine).
5. The developers and sponsors of the open source software used to create DCFI, including
   - DynamoRIO
   - Scintilla and SciTE
   - Python powerlaw
   - Google Protocol Buffers
   - the Ninja compiler for Windows
   - gcc/gdb
   - cmake
   - Ubuntu/Linux
   - VirtualBox
   - MinGW
   - Console2
   - Remmina
   - git
   - gnu binutils
   - octave
   - Inkscape
6. Microsoft DreamSpark for free licenses to Windows and Visual Studio.
7. The sponsors and maintainers of VirusShare ("because sharing is caring").
8. Albert Einstein for developing an effective approach to scientific modeling of invariants in the observable universe.
9. God for creating a universe worth observing, with goals that are worth the effort.
10. The NSF for sponsoring this research project.

This project was partly supported by the National Science Foundation under grant CNS-1228995.

ABSTRACT OF THE THESIS

DCFI: Control Flow Integrity for Modern Windows Applications

By

Byron Hawkins

Master of Science in Electrical and Computer Engineering

University of California, Irvine, 2014

Professor Brian Demsky, Chair

Control flow integrity or CFI has emerged as an important technique for preventing attacks on software. Previous approaches relied on static analysis and thus largely target static binaries and are limited in how tightly they can constrain a program's runtime behavior. Unfortunately, modern Windows applications make extensive use of dynamically generated code. We introduce a new dynamic analysis based approach in DCFI to control flow integrity that precisely learns a program's behavior by monitoring previous executions. DCFI is the first approach to demonstrate CFI in the presence of dynamic code generation and/or self-modifying code and is immune to recent variations on ROP attacks that thwart previous CFI approaches. DCFI underapproximates the legal executions of software applications and thus can potentially build tighter constraints than static approaches. As DCFI's knowledge of a program becomes more complete, it tightens its constraints on a program's execution, making successful attacks progressively more difficult.

We have implemented DCFI in DynamoRIO. Our experiences using DCFI indicate that it can protect modern desktop applications with dynamic code generation engines including the latest versions of Microsoft Word, Microsoft Excel, Microsoft PowerPoint, Microsoft Outlook, Google Chrome, and Adobe Acrobat. Experiments also show that DCFI effectively detects known exploits.

Chapter 1

Introduction

Malicious software or malware threatens nearly every computing environment. While defenses such as antivirus software, firewalls, and OS safeguards can reduce risks, many vulnerabilities remain. The security community has made significant progress in increasing the difficulty of common attacks: for code injections there is the $W \bigoplus X$ memory convention [25], for stack smashing there are canaries [13], and for buffer overflow attacks there is memory layout randomization [38]. But the problem of securing a networked computing environment remains open.

1.1 Control Flow Integrity

Researchers developed control flow integrity (CFI) to increase the difficulty of attacks that hijack program executions. The key insight of control flow integrity is that attacks often corrupt the program's control flow. Instead of identifying the particular kinds of corruption that may be useful to an adversary, control flow integrity aims to ensure the integrity of the intended execution. An ideal solution would constrain the program's execution strictly to

the paths that were intended by the developer and the development tools.

Previous research on CFI[1, 2] such as binCFI[49] and CCFIR[48] relies on a static analysis of a program to derive the expected behavior from the program's source or binary representations. The essential limitations that are common to these techniques are that (1) static analysis inherently introduces imprecision in the targets of indirect branches, (2) analysis of library calls is complicated by the fact that the components are typically linked dynamically at run time, (3) they may generate constraints on returns and/or gadget sizes that are too weak to effectively stop ROP attacks [9], and (4) they are unable to handle the dynamically generated code that modern Windows applications make extensive use of.

	Dynamically Generated Bindings				
	Gen	LM	T	BBs	JITs
Word	8	13	1284	16913	1
PowerPoint	9	15	2753	38105	1
Excel	4	9	432	2714	1
Outlook	4	9	440	3661	1
Chrome	6	10	796	17315	2
Adobe PDF	13	24	163	16965	2

Generators, **L**inked **M**odules, **T**rampolines, **B**asic **B**locks

Figure 1.1: Dynamic code generators in large Windows programs.

Recent versions of large Windows applications employ code generators in several modules. Figure 1.1 shows that not only do mainstream applications contain one or more JIT engines, it is also increasingly common for multiple modules to dynamically generate trampolines that link code at runtime. CFI techniques based on static analysis are unaware of these bindings, and these trampolines are typically not protected by W $\bigoplus$ X protections as they are written to executable memory pages. This vulnerability is critical for application security because dynamically generated trampolines most often bind application code to the core system services that malicious code uses to carry out its attacks.

This thesis introduces a new approach to control flow integrity that learns the expected

program behavior by observing executions, including dynamically generated trampolines and JIT code. We present DCFI, a process virtualization system that learns program control flow and evaluates the integrity of program executions on the basis of previously observed control flow behavior. The usage model for DCFI has two phases:

- DCFI monitors a sufficient number of program executions to learn the program's expected behavior.
- The alarm system is activated, and DCFI will raise an alarm when control flow violations exceed a threshold. DCFI can optionally continue learning even in this phase.

The key insight driving DCFI is that users should only execute code that is either familiar to them or those that they trust. This thesis makes the following contributions:

1. It presents DCFI, a process virtualization system that learns a program's expected control flow by monitoring its executions.
2. It presents a hierarchical constraint system for constraining a program's execution based on its previously observed behavior.
3. It presents a modular control flow integrity technique that significantly constrains both dynamically generated code and unverified use of potentially hazardous system calls.
4. It presents a permission system for constraining the relationship between JIT operations and the runtime behavior of dynamically generated code.
5. It evaluates DCFI on large, commonly used desktop applications including Microsoft Office 2013, the Chrome web browser, and Adobe Acrobat Reader, uses DCFI to detect several known exploits, and it measures the performance overhead of DCFI.

The thesis is organized as follows. Chapter 2 presents a typical usage scenario for DCFI. Chapter 3 presents the DCFI approach. Chapter 4 discusses how to configure DCFI's alarms. Chapter 5 extends the basic approach to handle dynamically generated code. Chapter 6 examines the DCFI defenses against current and hypothetical malware. Chapter 7 evaluates the effectiveness of DCFI on popular Windows programs, including the latest versions of Microsoft Office, the Chrome web browser, and Adobe Acrobat Reader. Chapter 8 discusses related work. We conclude in Chapter 9.

Chapter 2

Typical Usage Scenario

The goal of DCFI is to provide effective malware protection for modern computing environments. In this thesis we use as a motivating example the typical corporate software environment in which an IT department maintains control over the applications available to the employees, but DCFI is more broadly applicable. To protect corporate infrastructure from malware, the IT department configures computers to run applications under DCFI. Under this usage scenario, one approach for learning the expect control flow is for IT department to run the application under DCFI so that it can learn the expected control flow. In practice, there are other sources of training data available—many program of interest have millions of users and are extensively tested by their vendors before release. Binary analysis tools such as X-Force [35] could additionally be used in an automated training phase. These sources have the potential for providing extensive information about a program's ideal control flow. After the initial learning phase, the application is then released to the users with a constraint configuration that allows the employees to use the application comfortably, yet prevents most attacks.

In some training scenarios, the initial learning process may be incomplete for larger programs.

Instead of delaying the rollout, DCFI can enforce looser constraints on the application allowing the administrator to release it to the users immediately. While the application will be more vulnerable to attack under the looser constraints, DCFI will continue to learn the application's behaviors, and the learning rate will be much faster since observation data will be collected across all users. As DCFI's observations of the program's behavior become more complete, DCFI would progressively tighten its constraints. Note that for this reason, the effectiveness of DCFI improves with the number of users, since the size of an application's complete control flow graph is finite and its potential to exhibit new behaviors will gradually be exhausted.

Chapter 3

The DCFI Approach

Figure 3.1 presents an overview of the DCFI system. The system contains two primary components: (1) the client-side DCFI monitor, which observes program executions to detect potential attacks to the client machine and (2) the program signature server that collects, aggregates, and distributes information about expected client program behaviors across client machines.

DCFI monitors the execution of client programs to learn an initial set of constraints that characterize a program's typical control flow. Once DCFI has collected sufficient data about a client program, it monitors for divergences from the known control flow that may signify an attack. If DCFI detects an execution whose control flow violates the constraints, significantly diverging from known behavior, it raises an alarm, which may be configured to suspend or terminate the program.

DCFI extends the binary rewriting framework DynamoRIO[6] to efficiently monitor a client program's control flow. Client code is not executed directly from its original location in memory, but copied into a code cache with nominal instrumentation and executed from the cache. In most cases, DCFI minimizes overheads by only checking control flow integrity

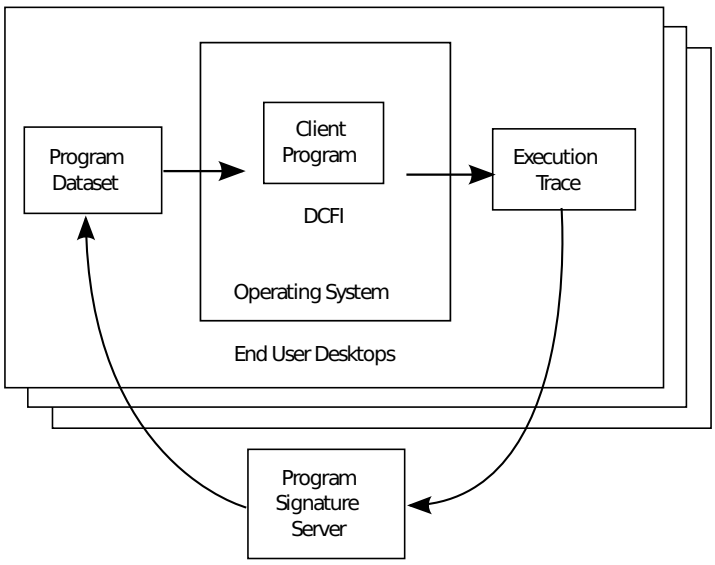

Figure 3.1: DCFI System Overview

constraints when code first enters the cache. One-time observations do not suffice for all cases, e.g. indirect branches and returns, and in those cases DCFI instruments the cached code to continuously monitor its control flow.

A centralized *program signature server* periodically collects traces from the client machines and compiles them into an updated program dataset by (1) merging all client observations together into a unified representation of the program control flow and (2) computing tighter constraints based on recent control flow divergences.

At program startup, DCFI loads a *program dataset* containing (1) the known control flow graph for the program and (2) a set of constraints on the execution of unrecognized control flow. DCFI updates a program dataset as it learns more about the client program's execution behavior, increasing the recognition of the program and lowering DCFI's tolerance for unrecognized execution behavior. Figure 3.2 presents DCFI's learning cycle.

A key benefit of DCFI is that it reverses the current advantage that malware developers have over older versions of a program. Popular security models concede malware development

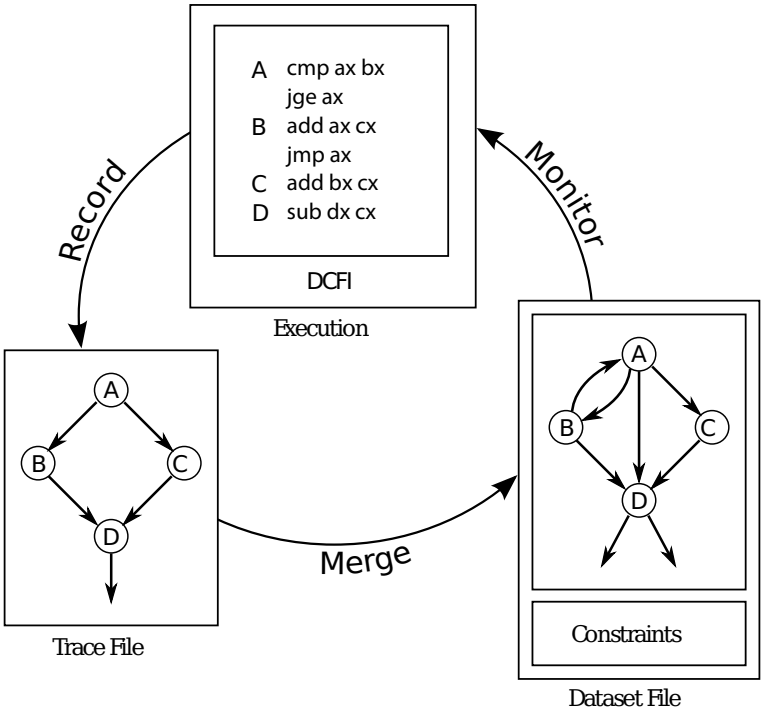

Figure 3.2: DCFI Learning Cycle

time, such that the longer a program is actively used, the more time a malicious coder has to develop an attack. The DCFI security model conversely improves security for a program the longer it is used—not only is the attack surface progressively reduced, but it also becomes a moving target. An attack that could potentially be successful in the first week of program usage may be consistently detected by DCFI after the second week, at which point the attack is obsolete.

Monitoring Program Executions: DCFI monitors program executions to dynamically construct a control flow graph or CFG. The CFG is initialized from the CFGs constructed by previous executions. Each node in the CFG represents a basic block of instructions that was executed by the client program, where a basic block in DynamoRIO is a contiguous sequence of instructions with one entrance and possibly many exits.

DCFI represents control flow transfers as CFG edges, so for example a `jmp` that represents

the i^{th} exit from block a to block b is represented as the directed edge $\langle a_i, b \rangle$. Each node is annotated with the physical location of its first instruction in memory, known as its *module-relative tag*, and a hash of its instructions. Edges are annotated with an enumerated type (such as *direct* or *indirect*). Function calls are modeled with two edges—one from the call site to the callee and one from the call site to the next instruction in the caller. When a thread makes a system call, DCFI does not monitor the execution of the thread into kernel code, but instead represents each system call as a node annotated with the system call number. Control transfers at `return` instructions are only recorded in the graph when they are unexpected, i.e., they return to a target other than the original call site. If a thread performs an unexpected return, DCFI creates a CFG edge for the control transfer.

Static Control Flow: The majority of CFG elements can be discovered by static analysis, including basic block hashcodes and direct branch targets. To avoid the difficulty of statically decoding a binary image, DCFI maintains a signed copy of each module and consults it whenever a new basic block or direct branch is observed during program execution. If the original image contains a matching control flow element, it is recognized just as if it had been observed in previous executions. It is normal for static control flow elements to diverge from the original binary image, for example in the common case of system call hooks that are installed at runtime, or for packed executables. To allow for these variations, DCFI maintains a version number for each basic block tag, which is incremented each time a new hashcode is observed at that tag. Since changing the target of a direct branch will also change the basic block hashcode, DCFI can require direct branches from the original instance of every basic block (per the binary image) to always have the original target address, yet can also recognize runtime modifications of those basic blocks without artificially constraining their direct branches.

Detecting Execution Anomalies: DCFI detects execution anomalies by monitoring the execution for deviations from the program's known runtime behavior as recorded in the

CFG. When the program first executes a new basic block a at address A, DCFI looks for a node with identifier A in the CFG. If such a node is found, the node's hash is compared with the instructions in block a. Different hashes for the same block indicate a potential violation of the execution's integrity. Control flow transfers are likewise compared against the corresponding CFG edges.

3.1 Anomaly Classification

A control flow anomaly does not necessarily indicate an attack. The CFG summarizes execution behaviors that DCFI previously observed and thus the CFG underapproximates a program's legal executions. While execution anomalies are expected to be rare for small programs or programs with extensive execution histories, executions of large programs without an extensive dataset may contain anomalies. Therefore, DCFI does not necessarily signal an alarm for every anomaly it detects.

Instead, DCFI classifies anomalies on the basis of low-level observations about the divergence from known control flow, and enforces constraints that tolerate a limited number of execution anomalies in each class. Constraints are established by the program signature server on the basis of (1) a user-specified false alarm tolerance, and (2) the frequency and size of anomalies occurring in each class during recent program execution. Chapter 4 elaborates the DCFI process for setting alarm thresholds in each anomaly class.

The key idea is to establish a hierarchy of anomaly classes ranging from specialized primitives that target the type of divergences that appear in attacks to more general primitives that include common divergences that can occur during normal execution.

1. Initially, DCFI strictly enforces constraints on attack primitives that occur rarely during execution of normal programs.

$$
\begin{array}{lcl}
\textit{expr} & ::= & \textit{true} \mid \textit{predicate} \mid \textit{expr} \ \wedge \ \textit{expr} \\
\textit{predicate} & ::= & \textit{bb-miss}(\textit{event}) \mid \textit{dbr-miss}(\textit{event}) \mid \\
 & & \textit{new-br}(\textit{event}, \textit{edge-type}) \mid \\
 & & \textit{new-inv}(\textit{event}, \textit{edge-type}) \mid \\
 & & \textit{intvl}(\textit{event}, \textit{expr}, \theta) \mid \\
 & & \textit{to-sys}(\textit{event}, \textit{expr}, n) \mid \textit{trmpl}(\textit{event}) \\
\textit{edge-type} & ::= & \textit{ind} \mid \textit{susp-ind} \mid \textit{ur} \mid \textit{fork} \\
 & & \textit{gen-perm} \mid \textit{gen-write} \\
\theta & ::= & \textit{time-constant} \\
\sigma & ::= & \texttt{tolerance parameter}
\end{array}
$$

Figure 3.3: Anomaly Classification Language

2. As DCFI continues to learn a program's behavior, it tightens constraints on the broader classes of anomalies until all aspects of a program's control flow are precisely constrained.

The driving observation behind anomaly classification is that the kind of control flow violations that are most useful to an adversary—and the most challenging to conventional defense techniques—occur infrequently in ordinary programs. Consider, for example, sophisticated attacks that use return oriented programming [37, 40, 8, 10] to build attacks using existing code. These attacks by necessity generate a large number of *unexpected returns*, returns to program locations that are inconsistent with the current call chain. To our knowledge, all the unexpected returns in our experiments are caused by inelegant handling of fiber context switches and call gates, and can be eliminated with simple improvements to DCFI. While there is no reason that any particular anomaly class *cannot* occur in a given program—it is possible to write a program in which any kind of anomaly occurs regularly—in practice it turns out that most ordinary programs rarely execute the attack primitives that are essential to today's malware.

To make our discussion about anomalies more precise, Figure 3.3 and Figure 3.4 present a language for classifying anomalies. An *expr* defines an anomaly class, such that each *event* observed by DCFI that matches *expr* a is classified as an instance of anomaly class a. A single *event* may match multiple *expr*s; i.e., anomaly classes are not always mutually

Predicate	Description
bb-miss	Basic block mismatch
dbr-miss	Direct branch mismatch in a memory mapped module image
new-br	Newly encountered branch of the specified *edge-type*
new-inv	Invocation of a new branch of the specified *edge-type*
intvl	CPU cycle count between occurrences of the specified *event*
to-sys	Invocation of a specific system call while the stack contains a frame in which *event* occurred
trmpl	New dynamically generated trampoline
Edge Type	
ind	Indirect branch, having its target selected/computed at runtime
susp-ind	New indirect branch with a non-exported target
ur	Unexpected return from a callee to a target other than the caller
gen-perm	Edge linking permission change to the region's generated code
gen-write	Edge linking a write to region of written code
fork	Edge linking the main module to a child process
θ	Time constant specifying a number of CPU cycles

Figure 3.4: Terms of the Anomaly Classification Language

exclusive. This language supports a broad spectrum of anomaly classes. For example, the general class *new-br*(*ind*) includes all indirect branch edges not found in the current dataset. Conversely, *to-sys*(*new-br*(*ind*), `0x66`) specifies an attack primitive in which one or more files are deleted from the filesystem while the current call stack has been influenced by an unexpected indirect branch target (i.e., in at least one live stack frame, an indirect branch was taken to a target that is new for that branch).

DCFI generates an *event* whenever it makes any of the following observations:

1. The execution includes a basic block for which: (1) there is no node in the dataset or (2) the dataset node's hash differs.

2. The execution takes a control flow branch for which there is no matching edge in the data set.

3. The execution changes `executable` memory permissions.

4. The execution writes to `executable` memory.

5. The execution makes a potentially hazardous system call.

6. The execution creates a child process.

Each *predicate* captures an observation that DCFI can make about an *event* on the basis of the execution history. While many such observations are possible, Chapter 6 explains how each predicate addresses a class of application vulnerabilities. Figure 3.4 provides an overview of each predicate. The time constant θ specifies a number of CPU cycles, as observed using the x86 `rdtsc` instruction. Special edge types *gen-perm* and *gen-write* indicate memory operations, as explained in Section 5.1, predicate *trmpl* constrains dynamically generated code, as discussed in Section 6.4, and predicate *fork* constrains the creation of child processes.

Anomaly Constraints: To establish constraints on anomalies in program execution, DCFI maintains a counter c_{expr} for each class of anomalies defined by *expr*. As the program executes, DCFI observes a sequence of *event* and increments the counter c_{expr} for each *expr*(*event*) that evaluates to *true*. The program dataset contains a set of thresholds τ_{expr}, and DCFI raises the alarm when any counter c_{expr} reaches corresponding threshold τ_{expr}. The remainder of this chapter describes implementation details of DCFI *event* observations, and Chapter 4 presents the methodology for DCFI alarm configuration.

3.2 Indirect Branch Caching

When the program executes an indirect branch from the code cache, the target will naturally be computed in the address space of the original application image, because the client application continues to execute all of its computations just as if it were not running in a code cache [7]. To keep the thread from returning to the native application code, an assembly routine in the code cache looks up the address of the cached copy in a hashtable, pausing the execution to cache a copy if none exists yet. This operation must be further interrupted

so that DCFI can generate an indirect branch event, not just for every new target block, but for every new pair of from and to addresses. To minimize interruptions, DCFI caches legal indirect control flow transfers and the assembly routine consults a second hashtable to determine whether the current from/to address pair has already been observed.

3.3 Return Instructions

In the common case when the execution's control flow at a `ret` instruction returns to the caller, there is no need to record this information as this was the expected behavior. To filter the corresponding edges from the CFG and prohibit `ret` statements from returning to the wrong caller, DCFI maintains a shadow stack of return addresses for each program thread. In the rare case of an unexpected target, a DCFI function is invoked to generate the unexpected return event and write it to the execution trace.

Nested Shadow Stacks: When a Windows program calls the OS API, the OS can perform a call back to the program *before* returning from the original API call. A fresh call stack is created for the callback. To distinguish the two contexts, DCFI pushes an empty shadow frame marked with a sentinel when it observes the instantiation of a callback stack, effectively creating a nested shadow stack.

Windows can invoke the callback from chain of tail calls such that the return from the callback goes directly to the program's original API call site. In this case, the sentinel value on the empty shadow stack frame is especially useful for keeping the shadow stack synchronized. DCFI simply unwinds to the frame below the sentinel and compares that frame with the current stack register to verify that the return is normal.

3.4 Modular API Preservation

Windows programs are comprised of many modules (both executables and DLLs). These modules can be shared between different programs and in many cases are distributed separately. Thus, DCFI modularizes the program's dataset to allow sharing individual module datasets between program datasets and to better handle updates to individual modules. In our experiments, we found that the applications that comprise Microsoft Office have more than half of their modules in common, allowing us to merge datasets for those modules across the different applications in Microsoft Office.

3.4.1 Exported Functions

One challenge in modularizing the dataset is to make each individual module dataset compatible with any other module dataset it may be linked against. For example, Word and Outlook use the latest Windows 7 version of `comctl32.dll`, while Excel and PowerPoint use that version together with the earlier Vista version (simultaneously). All Office programs share the core module `mso.dll` which has many CFG edges into `comctl32.dll`. In Word and Outlook, these edges always go to the Windows 7 version of `comctl32.dll`, but in Excel and PowerPoint, half of the edges go to the older Vista version. To accommodate this abstraction, DCFI splits the cross-module edges of the *program dataset* at the module boundary and annotates them with a hash representing the identifier of the API being invoked—usually the name of an exported function. For example, in the case of multiple `comctl32.dll` versions, a single cross-module edge from `mso.dll` represents an invocation of *either version* of the exported `comctl32.dll` function.

3.4.2 Callback Functions

Windows applications can use the callback programming pattern, which produces cross-module calls to private functions having no exported name. Since the function call is linked dynamically within the application at runtime, it would be overly restrictive for DCFI to associate the corresponding cross-module edge with the specific function that is invoked, since in the next execution of the same program could occur after an update and very reasonably link to a different implementation of the function. To preserve the generic properties of the callback mechanism, the cross-module edge is simply annotated with a hash representing (for example) `"<user32!callback>"`, where `user32.dll` is invoking the callback function. The effect on program evaluation is that DCFI will allow the callback site to target any function in any module that has ever been observed to receive any callback from `user32.dll`. While this evaluation is less strict than for cross-module calls to exported functions, it does constrain callback site while maintaining support for modularity.

3.5 Hash Canonicalization

To filter out location-dependent factors from the execution trace, canonicalization is applied to all relocatable operands while computing the node hashes. A relocatable operand is an absolute memory address that appears directly inside an assembly instruction. On disk, these operands point to the location of the target within the disk file, but the Windows loader is free to choose any physical location in memory for the module instance, causing the relocatable operand targets to be incorrect. The Windows loader fixes this by searching every module at load time for references to relocatable operands and replacing them with absolute locations. Simply hashing the absolute locations generated by the relocation process would cause DCFI to compute hashes that reflect the arbitrary locations of objects in memory, resulting in spurious hash mismatches between program executions, so DCFI canonicalizes

the instructions before computing the hash by un-relocating such operands.

Chapter 4

Alarm Configuration

In a real world deployment, each threshold τ_{expr} is calculated on the basis of (1) a user-defined false alarm tolerance and (2) the empirical distribution of c_{expr} observed in the most recent half of program execution history. The empirical distribution is only reliable when the number of executions in the dataset is a few times larger than the inverse of the desired user-defined false alarm rate; i.e., to establish a false alarm rate for the counter of 0.001% requires the dataset to contain on the order of 200,000 executions.

For large programs having complex control flow, early releases of the dataset may require such high thresholds on the more general predicates that little security is provided for those classes of anomalies. For example, if $\tau_{new\text{-}br(ind)}$ is calculated to be 200, DCFI might as well allow an infinite number of them. The program signature server will disable such a predicate until tighter convergence has been reached, since attacks can be designed to avoid exceeding the given threshold. Predicates on specific attack primitives occur rarely in normal programs, so in practice these will always be enabled by DCFI.

4.1 Statistical Modeling

The user may wish to set a lower false alarm rate than the length of the execution history allows, such that the calculated thresholds would be inaccurate. In this case DCFI uses statistical modeling to estimate the thresholds. The program signature server employs python package `powerlaw` [3] to generate a Power-Law or Log Normal distribution that best fits the empirical data for each c_{expr}, and then applies the generated model to calculate the corresponding τ_{expr}.

For small data sets, the thresholds for *new-br*(*ind*) and *new-br*(*susp-ind*)) have default minimum counts of 10 and 5, respectively, since lower thresholds are unrealistic for such general classes of anomalies. While an exploit could be devised to take control of the program in that number of branches, DCFI would continue to monitor execution, so the entire payload would have to execute within these minimum thresholds. The thresholds for *to-sys*(*ind*) are similarly cushioned with a minimum count of 10, excluding syscalls that could directly damage the filesystem, registry, environment, drivers or boot configuration (Figure A.1). In our experience, the tail portion of empirical data for each c_{expr} does not extend as far as the corresponding statistical model, resulting in a tendency to moderately over-estimate the required threshold τ_{expr}.

Chapter 5

Dynamically Generated Code

Many mainstream Windows programs dynamically generate code at runtime. Therefore, to provide effective control flow integrity for real applications, DCFI must recognize and constrain the dynamically generated code that is executed by a program. In many cases, different code is dynamically generated in different executions—making it infeasible to require that control flow matches for dynamically generated code. However, dynamic code generators produce code that interacts with the rest of the system through well-defined interfaces. Thus, DCFI can learn how the dynamically generated code interfaces with the rest of the system and later sandbox dynamically generated code to interact only via this interface. Moreover, DCFI can learn what parts of an application generate code and then later only allow execution of dynamic code produce by the application's dynamic code generator.

Dynamic code generation opens the opportunity for sophisticated attacks that do not modify static program code, but instead inject new code into dynamically generated code regions and then redirect the execution of at least one (possibly new) thread into the injection. When carefully implemented, the redirection can have a very small overlap with normal application code, possibly just one indirect branch. While it is easy for DCFI to detect that

the injection is executing dynamically generated code, this cannot be an immediate cause for alarm because most programs normally execute at least a few basic blocks of dynamically generated code.

In the programs we have observed—dynamically generated code can be classified into two distinct categories:

1. **Trampolines:** Small functions that bind with external components discovered at runtime. The number of distinct trampolines generated by any module is typically small enough to fit in the dataset.

2. **JIT Code:** Many applications incorporate JIT compilers to support internal web components and online optimization. Both require vast dynamic modules typically containing millions of basic blocks. JIT code typically never directly invokes system calls and all of the JIT entry points and exits are linked exclusively with the generating module.

The key difference between these two categories is that trampolines are typically conserved across different executions while JIT code frequently varies across different executions.

Throughout this thesis, we use the terms "JIT code" and "trampoline" to refer to dynamically generated code, and the term "generator" to refer to the pre-compiled module that generates JIT code and/or trampolines.

Dynamic Code Generation: There are commonalities between the tasks faced by both types of dynamic code generators. We can leverage three observations about the task of dynamically generating dynamic code of either kind to infer constraints on the code generators.

- The code generator allocates memory in which the dynamic code resides.

- A fixed set of call sites in the code generator modifies permissions on the pages containing dynamic code.
- A fixed set of store operations in the code generator writes code to its `executable` memory pages.

We next discuss the key differences between trampolines and JIT code.

Characteristics of Trampolines: The compact, predictable nature of trampolines makes it possible to effectively learn and enforce control flow graphs for them. Matching control flow graphs for trampolines is more complicated than for statically compiled modules as the addresses of nodes in the dynamically generated code graph are arbitrary and therefore cannot used to optimize the matching procedure. As trampoline code is executed, DCFI maintains a list of candidate subgraphs that match all the blocks and branches of the trampolines that have been observed so far. If at any point a trampoline executes a block or a branch that does not match any of the candidates in the dataset, DCFI increments the counter associated with predicate *trmpl*. For any subsequent nodes and edges executed in the unrecognized trampoline, DCFI will also increment the counters for the corresponding node and edge predicates.

Characteristics of JIT Code: The diversity of inputs to JIT compilers makes it infeasible to generate meaningful CFGs for the code they generate. Instead, DCFI represents the entire JIT code region with a singleton node, and provides security by using the existing predicates to learn the interfaces through which JIT code interacts with the rest of system and then to eventually sandbox the JIT code to these interfaces.

Differentiating JIT code from trampolines is straight-forward—JIT code regions grow without bounds and quickly become many orders of magnitude larger than even the biggest trampolines. Moreover, should some future system generate gigantic trampolines, it would be best for performance reasons to handle such trampolines using the JIT model.

5.1 Code Generation Constraints

As discussed earlier in this chapter, dynamic code generation in real world programs has a number of properties that we leverage to prevent attacks. We summarize these properties below:

- There are well defined entry points into dynamically generated code.
- There are well defined entries into the APIs that JIT code uses to interface with the rest of the system. For example, JIT code never directly performs system calls.
- There is a fixed set of basic blocks that is used to make regions of memory executable.
- There is a fixed set of stores that write all of the dynamic code.

DCFI leverages these properties as the basis of a permission system for dynamic code generation. DCFI learns or infers the constraints for a given application from previous executions and then uses the learnt constraints to enforce the permission system on future executions.

DCFI leverages the existing constraints on edges to constrain the entry and exits points to dynamic code as described in Properties 5.1 and 5.1. These constraints effectively sandbox dynamically generated code to interact with the rest of the system only via the well defined APIs that are exposed to the dynamic code.

Properties 5.1 and 5.1 control how new code can be generated. Enforcing these properties, for example, makes it impossible for an attacker to leverage a bug elsewhere in the system to write new code on top of dynamically generated code. To implement these properties, DCFI provides two important constraints on accesses to the memory pages on which dynamic code resides. When DCFI observes a program set the `executable` permission on any page of

memory, several sets of *gen-perm* edges are added to the execution trace, one for each module appearing on the call stack. These edges start from the basic block address of the most recent stack frame for each module, and end at each of the trampolines and/or JITs residing on that page of memory (see Section 5.3.2 for details). There will usually be very few such edges in the dataset, since most programs generate dynamic code in a very systematic way, resulting in DCFI quickly obtaining very tight constraints on the predicates *new-br*(*gen-perm*) and *new-inv*(*gen-perm*) (defined in Figure 3.3 and Figure 3.4).

Predicates *new-br*(*gen-write*) and *new-inv*(*gen-write*) provide similar constraints on writes to `executable` memory pages, effectively limiting the set of store instructions that can write to a specific trampoline or JIT code region while it is `executable`.

5.2 Monitoring Dynamic Code Generation

Monitoring all writes to executable memory by instrumenting store instructions is potentially very expensive, so instead DCFI leverages the virtual memory system to perform these checks by artificially revoking the `write` permissions on all dynamically allocated executable memory pages. When the client program writes to any of these pages, DCFI receives a page fault and looks in a shadow page table to see if the `readonly` permission is artificial or intended by the client program. In the artificial case, DCFI makes the page `writable`, executes the write on behalf of the client, and resets the page to `readonly`. DCFI generates a *gen-write* event at this time and writes the corresponding edges to the CFG. This approach is very efficient for most code generating applications, though for a basic block that very frequently writes to `executable` pages, DCFI will directly instrument the block for better performance.

5.3 Dynamic Code Events

To generate events for the execution of dynamic code, DCFI must:

1. Assign each basic block to a trampoline or JIT.
2. Correlate each trampoline or JIT to the respective subgraph or singleton in the dataset (if any).
3. Identify the basic blocks participating in each request for `executable` memory permission, and associate them with any dynamic code on those pages that gets executed.
4. Associate every basic block that writes to `executable` memory with the dynamic basic blocks that were written.

5.3.1 Dynamic Module Discovery

When a program is executed in DCFI with no prior dataset available, DCFI infers the role of each dynamic basic block. Each dynamic subgraph is initially designated as a trampoline. If any maximal contiguous subgraph of dynamic code exceeds 250 basic blocks in size, its generating module is designated as a JIT generator, and all of its dynamic code is replaced with a JIT singleton node. This ensures that DCFI can precisely constrain standard trampolines while avoiding the potential of incurring large overheads by attempting to match control flow graphs for JIT code regions or very large trampolines.

If a single module generates both trampolines and JIT code, DCFI simply designates all of its dynamic code as JIT. In our experiments this rule is never applied, because every JIT generator produces millions of nodes in a contiguous subgraph.

5.3.2 Shadow Page Table

To create the *gen-perm* and *gen-write* edges in the execution trace, and to evaluate any related alarm predicates, DCFI makes an association between observed memory operations and the execution of dynamic code. The lazy nature of the code cache makes this somewhat challenging, because at the time the memory operation occurs, the associated basic blocks will most often not be discovered yet. For example, when the application writes to `executable` memory, DCFI needs to create a *gen-write* edge from the basic block performing the write to the entry point of the new dynamic code. But the basic blocks for the new dynamic code do not yet exist in the code cache—in fact they do not exist in memory at all—because the write to `executable` memory is going to generate them. To simplify the implementation, DCFI waits for an application thread to enter the dynamic code before recording these edges:

DCFI maintains a shadow page table to track for each page: (1) the set of stores bb_{write} that write to `executable` memory pages and (2) the set of calls bb_{perm} that changed a page's permission to `executable`. When a thread first enters the dynamic code on page P, DCFI consults the shadow page table. If shadow page S_P contains bb_{write} entries, DCFI records a *gen-write* edge from each bb_{write} to that entry point of the trampoline or JIT on page P. If shadow page S_P contains bb_{perm} entries, DCFI records a *gen-perm* edge from each bb_{perm} to that entry point of the trampoline or JIT on page P. These edges are recorded during the initial discovery of each trampoline and JIT entry point on page P.

5.3.3 Distinguishing JIT Code from Trampolines

In general, ownership of dynamic code is assigned to the module that first enters it. This can be problematic if a module generates JIT code and is the first user of trampolines generated by other modules. In most cases the first entry into a trampoline is from the module that

generated it. But suppose module J generates JIT code and is also the first module to branch into a trampoline generated by module T. It would be possible for DCFI to mistake T's trampoline for J's JIT code, which would weaken the JIT constraints to include some trampoline entry and exit edges.

DCFI distinguishes this condition on the basis of the following refinement to the procedure for assigning ownership of dynamically generated code. Whenever JIT generator J branches to an unidentified dynamic block, DCFI checks the shadow page table to verify that some frame in J was on the call stack at the time the `executable` permission was granted to that code page. This check will fail in the case that J branches to T's trampoline, and DCFI accordingly leaves the identity of the dynamic block unassigned until an entry or exit connects it to a module that satisfies the ownership check.

Note that if the ownership of a trampoline is mistakenly assigned to the wrong module, the only consequence is that small trampoline regions may be duplicated in the CFG, resulting in better constraints with slightly worse performance.

Chapter 6

Attacks

We divide attacks that seek to inject malicious behavior into an execution into four general categories. We have designed DCFI's predicates to build selective constraints on the vulnerabilities that are frequently leveraged by these attacks to provide protection while DCFI is in the process of sufficiently learning the application's CFG such that it can eventually strictly enforce more general constraints. By using selective constraints on the execution of applications, DCFI can substantially increase the difficulty of common attacks while it completes the learning process.

6.1 Direct Code Modification

One simple way to attack an application is to directly inject malicious code into the existing binary code. This attack can also be initiated at runtime by dynamically modifying the in-memory image of a module. The predicates *bb-miss* and *dbr-miss* limit this attack by recognizing basic blocks and direct branches that do not match the dataset. These constraints are especially effective because they can also be applied to code outside the dataset by

consulting a signed copy of a module's binary file.

Normal application behavior often includes rewriting a few basic blocks, usually to install a hook in a system call trampoline or a common library function. The CFG encodes these runtime modifications with an annotation on the corresponding node, so that the application's code modifications can be distinguished from malicious ones. Since most hooks occur in central functions that are frequently used, the average and standard deviation for *bb-miss* and *dbr-miss* are always zero after the first run of all our benchmarks.

6.2 Return Oriented Programming

A more sophisticated attack leverages a program bug such as a buffer overflow to change the return addresses on the stack, effectively hijacking all the `ret` instructions in the unmodified code of the victim program. The DCFI predicate *new-br*(*ur*) limits this attack by specifically constraining the number of new unexpected return branches per program execution. As discussed above, in our experience the number of unexpected returns in a normal program execution is negligible. For those applications that may have more frequent unexpected returns, the predicate *new-inv*(*ur*) can additionally constrain the number of times each new unexpected return is invoked during the execution.

Recent conference publications have exposed vulnerabilities in the most advanced CFI techniques. Three generic attack primitives are demonstrated by [9] to defeat kBouncer [34] and ROPecker [12], among other CFI techniques. DCFI cannot be defeated by these primitives because:

1. The shadow stack detects any call that does not return to the subsequent instruction, and predicate *new-br*(*ur*) constrains the occurrence of such unexpected returns according to the frequency observed in the target program.

2. The DCFI predicates work equally well for detecting either a single contiguous widget or a disjoint series of widget fragments; the DCFI predicate counts will be exactly the same for either configuration of the same widget.

3. The DCFI dataset is never truncated or compressed.

In [39], specialized ROP attacks assembled from a minimal 32-bit C/C++ application (having only an empty `main()` function) are demonstrated to defeat kBouncer, ROPecker and ROPGuard [18]. By increasing the sophistication of the attack in [15], ROP gadgets built only from `kernel32.dll` can additionally defeat a collective implementation of all the aforementioned CFI techniques together with BinCFI [49]. These attacks are not able to defeat DCFI for the reasons mentioned above (note that the DCFI shadow stack does correctly distinguish a tail call from an unexpected return).

6.3 Indirect Branch Manipulation

Another popular malware target is the control flow branches for which the target is computed at runtime. This category accounts for about 30% of all branches in our benchmark applications. The compiler generates an *indirect branch* whenever multiple targets are possible, for example in the function table of a polymorphic class, or an invocation of a "listener" function. Indirect branches are vulnerable to attack because an adversary can hijack them by changing the target address at runtime, thereby redirecting the control flow. A common technique is to leverage a buffer overflow to overwrite an exception handler address on the stack [28], or to replace data objects with impostors having phony function tables [42, 21]. It can be difficult to determine whether an arbitrary indirect branch target represents correct or corrupt program behavior. Various attempts have been made to compute the correct set of branch targets by statically analyzing the code, but this approach has significant limi-

tations; while the information does of course exist, it is inconveniently spread out among numerous parties such as the compiler, the linker, the developer, and the runtime loader.

Attacks Comprised of Existing Widgets

Jump-oriented programming (JOP) [5] modifies several branch targets to construct a set of code widgets from unmodified basic blocks. The predicate *new-br*(*ind*) is often insufficient to prevent this type of attack, because ordinary executions of large programs with small training sets can often invoke a dozen or more new indirect branches (or new targets). Three predicates progressively make indirect branch constraints more selective, allowing for new indirect branches typically observed while still protecting against attacks.

1. *new-br*(*susp-ind*) selects for indirect branches with a target that is neither an exported function nor the target of any other indirect branch in the program dataset. This predicate leverages the observation that some strategies for building attack widgets transfer control to arbitrary locations in a function while structure entry points are by far the most common targets for indirect branches.

2. *new-br*(*susp-ind*)$\wedge$*new-inv*(*ind*) counts the number of invocations for each *new-br*(*susp-ind*). In our experiments (Chapter 7), roughly half the instances of new suspicious indirect branches were invoked only once or twice. Consequently DCFI sets a low threshold on this predicate, thereby preventing an attack from repeatedly using the same widget.

3. *intvl*(*new-inv*(*ind*), θ) counts invocations that occur at intervals shorter than θ. For *new-br*(*ind*) that were frequently invoked in our experiments, most invocations were separated in time by more than 10,000 CPU cycles, resulting in very low thresholds for $\tau_{intvl(new\text{-}inv(ind),10k)}$ and $\tau_{intvl(new\text{-}inv(ind),1k)}$. This constraint imposes a timing obligation on the adversary, who must find widgets that execute for a sufficiently long time before calling the next widget.

Single Branch Attacks

Alternatively, the adversary may focus the attack on a single indirect branch that leads to an especially useful function or system call [26, 30]. By corrupting the contents of an object that is the receiver object for a virtual method call later in the execution, the normal functionality of the program can be coerced into unintended operations—or intended operations at unintended times—such as leveraging support for running shell commands to format the hard drive. This attack relies on building an impostor of an object that is normally used for important system calls, which are then hijacked to carry out the attack. Since the adversary can corrupt one object to become an impostor of another object type, any error in the program can be a launch point for attack on any system call in the program. Detecting this kind of attack on the basis of the DCFI indirect branch predicates alone would require waiting for the dataset to be effectively 100% complete.

To focus constraints on the single-branch attack, the predicate *to-sys* observes the invocation count of each potentially hazardous system call that occurs on the substack following the specified *expr*. For example, the expression $\textit{to-sys}(\textit{new-inv}(\textit{ind}), s)$ will cause DCFI to raise system call suspicion every time a new indirect branch is invoked, and if any system call s is invoked before the stack drops below the suspicious frame, the counter $c_{\textit{to-sys}(\textit{new-inv}(\textit{ind}),s)}$ will be incremented. In our experience with common Windows applications, it is unusual for a program to execute potentially hazardous system calls on the substack after a new indirect branch. This selectivity makes it possible to set tight constraints on suspicious syscall predicates without false alarms.

6.4 Dynamic Code Generation Attacks

We next discuss how DCFI's predicates target attacks that inject malicious code into dynamically generated code regions. DCFI monitors indirect branches for transfers from precompiled code to dynamically generated code. When DCFI detects such a transfer, it consults the dataset to determine whether a previous execution branched into dynamically generated code from that basic block. If not, DCFI sets predicates *ind* and *susp-ind* to *true* for that edge and all subsequent edges in the dynamic code region. This limits the size of the injection to the counter threshold designated by the constraints on these predicates. If any system calls are made before the thread returns from the stack frame in which the *new-br*(*ind*) was observed, the counters for those system calls will also be incremented (see Section 5.1).

6.4.1 Trampoline Injection

If a thread enters injected malicious code from a basic block that normally enters a trampoline, DCFI will try to match the injection to the set of trampolines reachable from that block. DCFI will consider each of these trampolines to be a candidate match until the injection executes a basic block with a hash that does not match the corresponding trampoline block, where the correspondence is mapped by branch ordinal. The hash uniquely represents the set of instructions in the basic block (modulo absolute operands when the code is dynamically generated), so the match will fail if any instruction in the injection differs from an instruction in the corresponding trampoline block. DCFI discards a candidate when any of its basic blocks does not match the execution of the monitored thread. Likewise if the branches in the injection link the basic blocks together into a different structure than the trampoline, the edges will not match and DCFI will discard the candidate. If at any point there are no remaining candidates for the dynamic code, DCFI will consider the executing basic block

or edge a mismatch, and increment the counters for predicates *bb-miss* or *dbr-miss*, along with the counters for more specific predicates such as *new-br*(*ind*) and *new-br*(*susp-ind*).

6.4.2 Injection into JIT Code

To avoid detection by the DCFI predicates on basic blocks and edges, malicious code could potentially gain control through a basic block that normally enters a JIT code region (if the program has any), since that is the only category of code that DCFI cannot monitor at the level of basic blocks and branches. Note that to successfully inject code into a JIT region, the attacker must get the JIT to write the code (due to DCFI's constraints on which stores are allowed to generate code). Assuming an attack does this, the injected code will be free to use any instruction sequence without risk of triggering the DCFI predicates.

While DCFI cannot control the exact instruction sequence of such an attack, it does effectively sandbox the attack code. The key idea is to monitor the execution of dynamic code and match the exit edge from dynamic code to pre-compiled code (including the system call node). Such a "dynamic code exit" branch is monitored under the *dbr-miss* and *new-br*(*ind*) predicates, causing those counters to increment if the target block of the dynamic code exit has never been observed as the exit point from that particular JIT. In this way, these edge predicates constrain the code injection to using the normal API of the program's JIT code (if it has any). These same predicates constrain system calls because each system call is represented as an edge to a syscall singleton node.

6.5 Non-Control Data Attacks

Recent research demonstrates that many attacks on the control data of an application (e.g., ROP or JOP), can also be carried out through non-control data, either by influencing pro-

gram decisions at conditional branches and data-dependent indirect branches, or by modifying program state to change the effect of existing functionality [44]. In additions, the persistent malware proposed in Dynamic Hooks [45] restricts its direct influence to non-persistent data—data not reachable from any global variable—making it invisible both to hook detection mechanisms and coarse-grained CFI. Malware developers can discover vulnerabilities in the non-persistent, non-control data processing of an application using symbolic execution, making it a potentially significant threat in the near future.

The unique advantage of an attack on non-persistent, non-control data is that it can evade defenses that only observe a subset of application functionality. These defenses can only prevent an attack from getting started in certain specific ways, and are blind to many varieties of payloads. DCFI renders evasive strategies ineffective because every category of control flow in the vulnerable process—and all of its child processes—is subject to the DCFI constraints.

6.6 Incremental Attacks

In some deployments, DCFI may continually improve its program datasets based on observed program behaviors in the field. This deployment scenario opens up the threat of adversarial training in which an adversary constructs a sequence of attacks that incrementally adds the necessary edges to the CFG to implement the final full attack while remaining below the alarm threshold.

There is a straightforward countermeasure to such incremental attacks. At a high level, the purpose of updating the CFG with edges is to enable DCFI to tighten the alarm constraints to be more sensitive to potential attacks. Conceptually, a solution to adversarial training is to retain program data sets (comprised of both a CFG and the corresponding alarm

thresholds) for multiple different timestamps in the training process. DCFI would then sound an alarm if the alarm thresholds for any of the program data sets are exceeded. Thus an incremental attack would only be successful if initially performing the full attack would have been successful.

This can be efficiently implemented by observing that edges added later are likely to be infrequently traversed. Thus, we could use the current CFG implementation strategy to store the oldest CFG and then add a special type of edge with a timestamp to store the edges that are added in the updated datasets. DCFI could then efficiently maintain different versions of the alarm counters for each dataset.

Chapter 7

Evaluation

The goals of our evaluation are: (1) to evaluate the convergence of DCFI's predicates for common desktop applications, (2) verify that DCFI can successfully detect a variety of known exploits without raising false alarms, and (3) measure DCFI's overhead using the SPEC CPU 2006 benchmark suite. A small-scale simulation is expected to yield significantly worse convergence results than those observed in actual usage as we would expect real world usage to include orders of magnitude more observations.

We structure our evaluation into three sets of experiments:

- Predicate convergence experiments that measure whether the predicate counts converge for real world usage of large applications. The "real world" experiments (Section 7.1) measure the tightness of DCFI constraints while it is learning new behaviors. The procedure-based experiments (Section 7.2) measure predicate convergence for regular usage patterns in which DCFI observes every distinct application feature at least twice.
- Exploit experiments (Section 7.3) in which we enable the alarm system and evaluate how effectively DCFI can detect known exploits and how many false alarms we see for

normal executions.

- Performance experiments (Section 7.4) that measure the overhead of DCFI on the SPEC 2006 benchmarks.

We conduct the experiments in Windows 7 SP 1 running in VirtualBox 4.2.10 on an Intel Xeon E3-1245 v3 CPU. The Windows Update service and application updates are disabled to maintain consistency throughout the experiments. The host operating system for VirtualBox is Ubuntu 13.04. The experiments focus on 6 commonly used Windows programs:

1. Microsoft Office 2013 (Word, PowerPoint, Excel, Outlook)
2. Google Chrome (web install from December 2013)
3. Adobe PDF Reader XI (web install)

7.1 Eating Our Own Dog Food

In the following experiments we did not follow a specific procedure for using the applications, but simply performed our normal daily tasks with them over the course of 2 or 3 weeks. The program dataset was regularly updated until the last 5 days of the experiment, at which point the data set was frozen and the trial began. Since there were no fixed bounds on application usage or input content, we expected to exercise some features during the trial that were not covered by the training. To demonstrate the safety of DCFI while discovering reasonable amounts of new code, the reporting for these experiments focuses on predicates that discern a degree of risk from unrecognized indirect branches.

7.1.1 Everyday Email with Outlook

The author configured Outlook with IMAP access to his personal email account and POP access to his university gmail account. Throughout the course of the training and trial periods, he viewed every incoming message, allowing it to load all images along with peripheral content such as conversation history and related posts from Facebook and LinkedIn. He also created filters to sort incoming messages among 25 folders.

Only a handful of new indirect branches are encountered during the test, but considering that Outlook is connected to the Internet and receiving large amounts of untrusted content, it is very possible for a well-crafted exploit to succeed without producing a larger divergence from expected control flow. For this reason, predicate *to-sys* observes syscalls invoked while the call stack contains a live frame in which any new indirect branch was executed. Three memory related syscalls occur in this context: `NtMapViewOfSection`, `NtProtectVirtualMemory` and `NtAllocateVirtualMemory`.

While these could be used in a code injection attack, they are relatively safe while DCFI is using this Outlook dataset because there is no JIT. If an exploit were to begin by mapping packed malware to memory and setting it executable, DCFI would assume any entry point to the injection could only reach a trampoline, and therefore DCFI would enforce the stricter trampoline constraints.

7.1.2 Browsing with Google Chrome

To build up the program dataset, the author performed all research and personal browsing tasks in Chrome under DCFI, including online shopping, reading news, social networking, watching videos, googling for technical information, reading related works, and downloading software components. To broaden the range of the experiment, he selected several

feature-intensive websites, including the top 10 best Flash and HTML 5 websites reported by ebizmba.com, the demo page of JavaScript 3D and WebGL toolkit threejs.org, and many sites referenced by backbonejs.org. He especially enjoyed watching the Super Bowl live via Flash video, and followed Facebook posts about the game, sent SMS via Google Voice to fellow fans, and followed live game stats on cbssports.com.

During the 5 day trial period, he returned to every site visited during training, additionally browsing similar sites that had been specifically avoided during training. To stress the DCFI dataset, he visited script-intensive sites featuring JavaScript art, such as (1) a videosphere [19], which renders several videos simultaneously on 3D tiles that form the wobbly surface of a sphere and (2) a textured video [4], which continuously integrates a triangular distance texture into each frame of a video.

In a few executions, DCFI observed the combination of memory mapping and setting memory permissions, which in the context of Chrome is much more dangerous than Outlook because there are two JIT areas that DCFI cannot protect with basic block and branch predicates. Most of these suspicious executions additionally had one *new-br*(*gen-perm*)—if a code injection caused this variation in memory handling, it could have feasibly replaced a JIT entry point with its injection. But these executions contain no other dangerous system calls under stack suspicion, indicating a successful exploit would have to return from the manipulated branch without further anomaly, and then rely on the normal behavior of Chrome to carry out the damaging system calls.

7.1.3 Writing This Paper

The author of this thesis edited the source `latex` files in SciTE running in DCFI, and then compiled them to PDF under DCFI using MikTek `pdflatex`. The training period included the first draft of Chapters 1-5 of the thesis, along with implementation of the

DCFI predicates *gen-perm* and *gen-write*, and the post-processing tools that generated the charts and tables.

Memory management syscalls occur in SciTE during a few executions, though it is relatively safe because the SciTE dataset only contains a single dynamic trampoline having 3 basic blocks. This offers very little slack for a code injection to operate without setting off the DCFI alarm. A few new indirect branches are encountered in several runs of pdflatex, usually caused by layout variations or the addition of a new package. But this dataset contains no dynamic code, which means that in real usage DCFI would raise an alarm for any code injection. Moreover, no dangerous system calls are ever made under the direct influence of the new indirect branches.

7.2 Convergence for Large Applications

These five experiments simulate typical office application usage of viewing, editing, and creating documents. We have designed our experiments to simulate the diverse usage that office applications receive in the real world. Our experiments are structures as a set of sessions that explore new application usage scenarios. Each session took 2 to 5 minutes, and was repeated 20 to 100 times. We started each experiment with a baseline data set built by using the application generically, but not for the prescribed usage scenario. We updated the program dataset after each session in the training portion of each experiment, and then stopped updating the dataset for the test portion of the experiment. Figure 7.1 reports average predicate counts for the first half of the training set, the complete training set, and the test set.

In each session, we performed ordinary tasks such as creating and editing documents. We tried to simulate the diversity of real world usage by performing tasks in the many different

	Word Create			PowerPoint Create			Excel Create		
Predicate	$\frac{1}{2}$ Training	All Training	Test	$\frac{1}{2}$ Training	All Training	Test	$\frac{1}{2}$ Training	All Training	Test
new-br(*ind*)	2k ± 2k	21.0 ± 13.3	8.90 ± 1.07	2k ± 1k	149 ± 91	13.8 ± 5.7	589 ± 359	183 ± 119	7.07 ± 2.97
new-br(*susp-ind*)	393 ± 352	6.90 ± 3.37	1.50 ± 0.56	639 ± 544	37.9 ± 21.9	4.00 ± 1.46	156 ± 90	44.6 ± 26.5	2.47 ± 1.16
new-inv(*susp-ind*)	29k ± 29k	22.4 ± 11.8	2.90 ± 1.38	1k ± 1k	924 ± 832	7.88 ± 3.63	820 ± 691	858 ± 661	25.1 ± 17.0
new-br(*ur*)	0.60 ± 0.60	0.10 ± 0.10	0.00 ± 0.00	1.33 ± 1.21	0.20 ± 0.18	0.88 ± 0.88	1.43 ± 1.43	0.14 ± 0.13	0.00 ± 0.00

Figure 7.1: Predicate convergence in procedure-based experiments (lower is safer)

ways that the programs support. For example, we used a variety of interfaces such as the hover menu, the ribbon menu, and shortcut keys.

7.2.1 Creating Microsoft Word Documents

For this experiment, we created 10 new documents in the training set and 10 new documents in the test set. In each document, we replicated the first page of a (distinct) research paper randomly chosen from the 2013 USENIX Security Symposium. The replica included all formatting except figures and references, and we uploaded each file to a SkyDrive account through the "Save As..." dialog (which internally is a complete instance of Windows Explorer).

DCFI learns the new functionality from the SkyDrive connector in the first 5 iterations. Column editing was also new to this dataset, and only 8 runs were needed to reach effective coverage.

At this level of convergence, the most permissive predicate is *new-br*(*susp-ind*), which could allow up to 8 suspicious indirect branches (20 indirects overall), with fewer than 50 total invocations (i.e., not per branch). It could be challenging to implement an effective payload under these constraints.

7.2.2 Creating PowerPoint Presentations

For this experiment, we created 25 short presentations about software security in the training set and 25 in the test set. Each presentation started with a template downloaded through PowerPoint's HTML browser, we then created a title slide and 3-4 additional slides with 2-8 bullet points. After adding a conclusion slide featuring an image downloaded through PowerPoint's HTML-based online image search, the file was saved to the local disk.

Four modules produced significant new code that required 25 executions to reach coverage, partly because of user variations in creating and manipulating bullet points.

7.2.3 Creating Excel Worksheets

For this experiment, we created 15 Excel worksheets in the training set and 15 in the test set. Each worksheet replicated the crime data reported by the FBI in 2012 for a different state. Population and crime count data were manually entered, and formulas were inserted to calculate the crime rate per category and the state totals. We roughly replicated the formatting from the web page and saved each worksheet to the local disk.

Most of the *new-br*(*susp-ind*) occurred in 4 runs for which the input data caused a "divide by zero" error in the cell formula for the crime rate. Some of worksheets had roundoff errors, and thus like real world usage we improvised new and different formulas that avoided these errors.

7.2.4 Viewing PDFs with Adobe Reader

To establish basic coverage of the broad range of formats and graphics found in PDF files, a screen robot paged through 5,000 randomly downloaded documents, uploading them to ac-

robat.com through the reader's Flash-based file browser. In the test we viewed and uploaded 100 (distinct) manuals for personal security products.

During the test set, the sole anomaly we encountered was 17 suspicious indirect branches. Our predicates maintains effective constraints even under this anomaly as only 11 distinct syscalls were ever invoked under new indirect branches in this experiment—all benign in the observed combination.

7.2.5 Giving PowerPoint Presentations

A screen robot trained DCFI for PowerPoint by watching 2,000 randomly downloaded presentations. To test convergence, we viewed 100 (distinct) nutrition and health lectures from the FDA website.

The dataset converges substantially over the course of the training, with an average of 100 new indirect branches in the first half of training reduced to an average of just one during the test runs. Out of all test executions, even the significant outlier encountered just 6 suspicious indirect branches.

7.3 Exploits

Four exploit experiments demonstrate that DCFI does not raise false alarms during normal program execution, but consistently detects exploits—raising the alarm before the payload is executed. We were unable to find reliable exploits for the recent versions of our office applications from the previous section. This is likely because vendors pay substantial bounties that effectively keep exploits off the market [27, 41]. Instead, we chose a representative set of published exploits [29, 32, 31] that cover a variety of application types and vulnerabilities.

We trained DCFI to recognize normal execution of each vulnerable program in two phases: the first to establish coverage of a substantial feature set, and the second to calculate effective alarm thresholds. We used the statistical modeling method because the datasets were too small for calculating thresholds at our target false alarm rate of (empirically) zero.

7.3.1 OSVDB-ID 104062 · Notepad++

We trained DCFI to recognize Notepad++ and the CCompletion plugin during development of a 500-line graphical chess game. After enabling the alarm, we continued adding features for one hour with zero false alarms. Then we opened the published exploit file, selected the text and invoked the CCompletion "locate identifier" function, which attempts to exploit a buffer overflow. DCFI raised the alarm on the *unexpected return* predicate as the exploit attempted to gain control of `eip`.

7.3.2 OSVDB-ID 93465 · Adrenalin Player

We trained DCFI to recognize the Adrenalin multimedia player by opening and modifying dozens of playlists, and playing 100 mp3 files. After enabling the alarm, we continued similar usage of the player for two hours with zero false alarms. Then we opened the published exploit playlist, which leverages a buffer overflow to initiate an ROP widget that forks an arbitrary child process. DCFI raised the alarm on the *new-br*(*ur*) predicate when execution first entered the chain of bogus return addresses.

7.3.3 CVE-2014-1610 · MediaWiki

We trained DCFI to recognize Microsoft IISExpress running MediaWiki by creating and editing pages, creating new users, and uploading files. The published exploit leverages unsanitized input to manipulate a DOS command into executing a malicious subexpression, which has the side effect of sending malformed input to the ImageMagick program (the original callee of the command). We wanted to demonstrate that DCFI can detect the specific actions of an exploit, not just error handling anomalies caused by a side effect, so we deliberately sent a variety of bad `http` parameters to the thumbnail script during training. After enabling the alarm, we continued posting on the wiki for one hour—also sending more incorrect (but benign) inputs to the thumbnailer—all with no false alarms. Then we invoked the published exploit. Since it only manipulates non-control data, it had no effect on the control flow of the intermediate IIS and PHP processes. DCFI raised the alarm on the first instance of predicate $\mathit{intvl}(\mathit{new\text{-}br}(\mathit{ind}), 10^3)$ in the spawned DOS shell.

To evaluate the resilience of the DCFI constraints, we artificially raised all indirect branch thresholds for the DOS shell, allowing the hostile command to execute. Then we challenged successive layers of the DCFI constraint hierarchy:

1. First, the exploited DOS command attempted to download a file using the Windows utility bitsadmin. DCFI raised the alarm on predicate $\mathit{new\text{-}br}(\mathit{fork})$ because the DOS dataset contained no *fork* edge to a process named bitsadmin.

2. After artificially raising the threshold for $\mathit{new\text{-}br}(\mathit{fork})$ in the DOS dataset, we repeated the exploit. DCFI raised the alarm because there is no DCFI dataset for the bitsadmin program in the MediaWiki installation.

3. Finally we extended the exploit command to first copy the bitsadmin program to a temp directory and rename it to gswin32, which is commonly invoked by PHP via

the DOS shell. DCFI raised the alarm within the (renamed) bitsadmin process on the *new-br*(*ind*) predicate, because the observed control flow did not match the DCFI dataset for gswin32.

DCFI was configured to throw an exception on alarm, so IIS terminated the attack threads gracefully, allowing Media-Wiki service to continue despite the exploit attempts.

7.3.4 CVE-2006-2465 · mp3info

A script trained DCFI to recognize the mp3info utility by executing 2,000 randomly formulated view and edit commands on 300 mp3 files. After enabling the DCFI alarm and adding 50 more mp3 files, the script executed 500 similar commands with no false alarms. Then the script executed the published exploit, in which a specially crafted command leverages a buffer overflow to pivot the stack into an ROP chain. DCFI observed the phony return address and raised the alarm on the *new-br*(*ur*) predicate.

7.4 Performance

We evaluated the performance of DCFI relative to native execution speed on the SPEC CPU 2006 benchmark suite [20]. SPEC CPU 2006 consists of a diverse set of CPU bound applications across several different application domains and languages—SPEC CPU 2006 contains 7 C++ applications, 12 C applications, 4 C/Fortran applications, and 6 Fortran applications.

We measured a geometric mean of 25.5% slowdown across all of the benchmarks in SPEC. Figure 7.2 presents the individual overheads. In our experience, our current prototype of DCFI on average doubles DynamoRIO's instrumentation overhead for indirect branches.

Performance has not been our focus for DCFI, and a number of opportunities remain to further reduce overhead. Our current implementation of DCFI disables DynamoRIO's trace cache for ease of implementation. Reenabling the trace cache would eliminate DCFI's indirect branch instrumentation overhead in many cases and based on experiments with DynamoRIO would likely significantly reduce the overheads of DCFI on the SPEC benchmarks.

Performance Overhead of DCFI									
400.perlbench	2.03	401.bzip2	1.22	403.gcc	1.61	410.bwaves	0.97	416.gamess	1.22
429.mcf	1.05	433.milc	1.00	434.zeusmp	1.06	435.gromacs	1.14	436.cactusADM	1.05
437.leslie3d	0.97	444.namd	1.01	445.gobmk	1.50	447.dealII	1.27	450.soplex	1.12
453.povray	2.40	454.calculix	1.11	456.hmmer	1.03	458.sjeng	1.88	459.GemsFDTD	0.96
462.libquantum	1.01	464.h264ref	1.33	465.tonto	1.33	470.lbm	0.92	471.omnetpp	1.49
473.astar	1.04	481.wrf	1.11	482.sphinx3	1.21	483.xalancbmk	3.17	**Geometric mean**	**1.255**

Figure 7.2: Normalized DCFI Execution Times for Spec CPU 2006

Measuring the performance overhead of DCFI on our larger Windows applications is challenging because they are interactive and it is not clear exactly what to time. However, we report our subjective experiences using these applications. The user experience of DCFI for a small program like SciTE is not distinguishable from native execution, even while quickly editing many large source files before a paper deadline. More complex programs like Word take a few seconds longer to start, and certain features take a couple seconds to load (e.g., the built-in image search), but the user interface is otherwise responsive and natural. Programs that generate significant amounts of dynamic code have similar delays in DCFI, as is typical of dynamic binary translation.

Chapter 8

Related Work

DCFI is distinguished from previous CFI techniques by its dynamic approach to discovering the target program's normal control flow. The key advantages of DCFI relative to other approaches [46] are that it is able to handle self-modifying code, dynamic code generation, and can potentially more precisely enforce control flow as it is not constrained by fundamental limitations of static analysis.

The *vtable* protection tool SafeDispatch[22] instruments C++ object function tables with a dynamic check that ensures only valid targets in the class hierarchy are called. Other kinds of control flow branches are not secured. Within the domain of *vtable* pointers, DCFI can provide stronger constraints by discovering at runtime which targets are actually called from each individual branch. Other limitations of SafeDispatch are that (1) it will instrument base classes in shared libraries to reject subclass targets in any library that was not available at the time the tool was applied and (2) it requires source code that may not be available to the end user.

A similar work (unnamed) [43] assembles *vtable* data at runtime to avoid the former limitation. It performs best among major CFI techniques, but only protects forward edges in the

CFG, and also requires source code to be available.

CCFIR [48] uses a static rewriter to provide CFI with extremely low overheads. CCFIR does not support protecting applications with dynamic code generation or self-modifying code and hence DCFI is more broadly applicable.

Several approaches target specific attacks [16, 11, 33], such as ROP, but may not be as robust as CFI to new types of attacks.

BinCFI also instruments COTS binaries, inserting dynamic bounds checks for indirect branches and routing them through auxiliary trampoline tables[49]. The set of allowed branch targets is approximated on the basis of developer and compiler conventions, resulting again in looser constraints than the DCFI predicates can specify. When applied to an unusual binary, for example having hand-coded assembly, the instrumentation will either skip over the unconventional code, or will instrument it incorrectly and cause program errors. The DCFI predicates are fully compatible with any executable code, including dynamically generated code that cannot be instrumented on the basis of static analysis even at runtime.

Clearview [36] uses learning to patch software errors. Clearview uses Daikon to learn constraints on variables, identifies violations of these invariants on erroneous executions, and generates patches the restore the invariants. While DCFI uses similar techniques to Clearview, a key difference is that Clearview focuses on generating repairs (and relies on external mechanisms to detect erroneous executions) while DCFI focuses on detecting attacks.

XFI is a static rewriter-based approach to CFI [17]. It checks coarser grained constraints on control flow that DCFI, cannot handle hand-coded modules, and cannot handle the dynamically generated code that appears in many modern applications. MoCFI seeks to enforce CFI on smartphones [14]. It uses static analysis to extract the CFG and simply enforces that statically unresolvable indirect jumps target a function entrance. DCFI can enforce more precise constraints on control flow. Program shepherding enforces various

execution policies such code origins on program executions [23]. In general it enforces weaker properties than CFI systems.

PittSField [24] and other SFI [47] tools can verify that assembly code with certain properties maintains type safety. SFI in general provides stronger guarantees than CFI, but is only applicable to specially generated code.

Chapter 9

Conclusion

We presented DCFI, the first tool for enforcing control flow integrity based on previous observations of a program's runtime behavior. DCFI can effectively handle dynamic code generation by inferring and then enforcing a permission system on dynamic code. Our experience indicates that DCFI is able to effectively enforce constraints on real-world programs that would make effective attacks more challenging.

Bibliography

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

[2] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information and System Security*, 13(1), October 2009.

[3] J. Alstott, E. Bullmore, and D. Plenz. powerlaw: A python package for analysis of heavy-tailed distributions. *PLoS ONE*, 9(1):e85777, 01 2014.

[4] beginfill.com. beginfill.com homepage. `beginfill.com/WebGL_Video3D`, 2014.

[5] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang. Jump-oriented programming: A new class of code-reuse attack. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, ASIACCS, 2011.

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

[7] D. Bruening, Q. Zhao, and S. Amarasinghe. Transparent dynamic instrumentation. In *Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments*, VEE, 2012.

[8] E. Buchanan, R. Roemer, H. Shacham, and S. Savage. When good instructions go bad: Generalizing return-oriented programming to RISC. In *Proceedings of the 15th ACM Conference on Computer and Communications Security*, CCS, 2008.

[9] N. Carlini and D. Wagner. ROP is still dangerous: Breaking modern defenses. In *23rd USENIX Security Symposium*, August 2014.

[10] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy. Return-oriented programming without returns. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, CCS, 2010.

[11] P. Chen, H. Xiao, X. Shen, X. Yin, B. Mao, and L. Xie. DROP: Detecting return-oriented programming malicious code. In *Proceedings of the 5th International Conference on Information Systems Security*, ICISS, 2009.

[12] Y. Cheng, Z. Zhou, M. Yu, X. Ding, and R. H. Deng. ROPecker: A generic and practical approach for defending against ROP attack. In *Proceedings of the ISOC Network and Distributed System Security Symposium (NDSS '14)*, 2014.

[13] C. Cowan, C. Pu, D. Maier, H. Hintony, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the USENIX Security Symposium*, USENIX Security, 1998.

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

[15] L. Davi, A.-R. Sadeghi, D. Lehmann, and F. Monrose. Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection. In *23rd USENIX Security Symposium*, August 2014.

[16] L. Davi, A.-R. Sadeghi, and M. Winandy. ROPdefender: A detection tool to defend against return-oriented programming attacks. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, ASIACCS, 2011.

[17] U. Erlingsson, S. Valley, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula. XFI: Software guards for system address spaces. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*, OSDI, 2006.

[18] I. Fratric. Runtime prevention of return-oriented programming attacks. http://ropguard.googlecode.com/svn/trunk/doc/ropguard.pdf, 2012.

[19] Goo Labs. Goo Labs homepage. `labs.gooengine.com/videosphere`, 2014.

[20] J. L. Henning. SPEC CPU2006 benchmark descriptions. *SIGARCH Computer Architecture News*, 34(4), September 2006.

[21] G. Hoglund. Advanced buffer overflow techniques. `www.blackhat.com/presentations/bh-asia-00/greg/greg-asia-00-stalking.ppt`, 2000. Black Hat Asia.

[22] 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, 2014.

[23] V. Kiriansky, D. Bruening, and S. Amarasinghe. Secure execution via program shepherding. In *Proceedings of the 11th USENIX Security Symposium*, USENIX Security, 2002.

[24] S. McCamant and G. Morrisett. Evaluating SFI for a CISC architecture. In *Proceedings of the 15th USENIX Security Symposium*, USENIX Security, 2006.

[25] 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`.

[26] Microsoft. Vulnerability in Internet Explorer could allow remote code execution. `technet.microsoft.com/en-us/security/advisory/961051`, 2008.

[27] Microsoft. Microsoft bounty programs. `msdn.microsoft.com/en-us/library/dn425036.aspx`, 2013.

[28] Microsoft. How to enable SEHOP in Windows. `support.microsoft.com/kb/956607/en-US`, 2014.

[29] Mitre. CVE database. `cve.mitre.org/`, 2014.

[30] H. D. Moore. Microsoft Internet Explorer data binding memory corruption. `packetstormsecurity.com/files/86162/`, 2010.

[31] Offensive Security. Exploit database. `www.exploit-db.com/`, 2014.

[32] OSVDB. Open sourced vulnerability database. `osvdb.com/`, 2014.

[33] 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, 2012.

[34] V. Pappas, M. Polychronakis, and A. D. Keromytis. Transparent ROP exploit mitigation using indirect branch tracing. In *Proceedings of the 22nd USENIX Conference on Security*, SEC, 2013.

[35] F. Peng, Z. Deng, X. Zhang, D. Xu, Z. Lin, and Z. Su. X-Force: Force-executing binary programs for security applications. In *23rd USENIX Security Symposium*, San Diego, CA, Aug. 2014. USENIX Association.

[36] 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 *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP, 2009.

[37] R. Roemer, E. Buchanan, H. Shacham, and S. Savage. Return-oriented programming: Systems, languages and applications. *ACM Transactions on Information and System Security*, 15(1), March 2012.

[38] D. D. Sandeep Bhatkar and R. Sekar. Address obfuscation: An efficient approach to combat a broad range of memory error exploits. In *Proceedings of the 12th USENIX Security Symposium*, USENIX Security, 2003.

[39] F. Schuster, T. Tendyck, J. Pewny, A. Maa, M. Steegmanns, M. Contag, and T. Holz. Evaluating the effectiveness of current antiROP defenses. In *Research in Attacks, Intrusions, and Defenses - 17th International Symposium, RAID 2014*. Springer, 2014.

[40] H. Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS, 2007.

[41] M. Sparks. Google offers $2.7m bounty to hackers. `www.telegraph.co.uk/technology/google/10601907/Google-offers-2.7m-bounty-to-hackers.html`, 2014.

[42] L. Szekeres, M. Payer, T. Wei, and D. Song. Sok: Eternal war in memory. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, 2013.

[43] 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 *23rd USENIX Security Symposium*, Security, San Diego, CA, Aug. 2014. USENIX Association.

[44] USENIX. *Non-Control-Data Attacks Are Realistic Threats*. USENIX, August 2005. Source code of attack programs can be obtained from the attached txt file.

[45] S. Vogl, R. Gawlik, B. Garmany, T. Kittel, J. Pfoh, C. Eckert, and T. Holz. Dynamic hooks: Hiding control flow changes within non-control data. In *23rd USENIX Security Symposium*, San Diego, CA, Aug. 2014. USENIX Association.

[46] Y. Xia, Y. Liu, H. Chen, and B. Zang. CFIMon: Detecting violation of control flow integrity using performance counters. In *Proceedings of the 2012 Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN, 2012.

[47] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Orm, S. Okasaka, N. Narula, N. Fullagar, and G. Inc. Native client: A sandbox for portable, untrusted x86 native code. In *Proceedings of the 2007 IEEE Symposium on Security and Privacy*, SP, 2009.

[48] 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*, SP, 2013.

[49] M. Zhang and R. Sekar. Control flow integrity for COTS binaries. In *USENIX Security Symposium*, USENIX Security, 2013.

Appendix A

High-Risk System Calls	
NtAddBootEntry	ZwRestoreKey
ZwAddDriverEntry	NtResumeProcess
ZwAlertResumeThread	ZwResumeThread
ZwCreateFile	NtSaveKey{,Ex}
ZwCreateKey	NtSetBootEntryOrder
ZwCreateProcess{,Ex}	ZwSetBootOptions
ZwCreateThread{,Ex}	ZwSetDriverEntryOrder
NtCreateUserProcess	NtSetEaFile
ZwDeleteBootEntry	NtSetInformationFile
ZwDeleteDriverEntry	ZwSetInformationProcess
ZwDeleteFile	NtSetSecurityObject
NtDeleteKey	NtSetSystemEnvironmentValue{,Ex}
ZwDeleteValueKey	NtSetSystemInformation
ZwDeviceIoControlFile	ZwSetSystemPowerState
ZwFsControlFile	ZwSetSystemTime
ZwLoadKey{,2,Ex}	ZwSetValueKey
ZwModifyBootEntry	ZwShutdownSystem
ZwModifyDriverEntry	NtSuspendProcess
ZwOpenProcess	NtSuspendThread
ZwProtectVirtualMemory	NtTerminateProcess
ZwRaiseException	ZwUnloadDriver
ZwRaiseHardError	ZwUnloadKey{,2,Ex}
ZwRenameKey	ZwWriteFile
ZwReplaceKey	NtWriteFileGather

Figure A.1: Syscalls for which DCFI sets the minimum $\tau_{to\text{-}sys(ind)}$ to zero because a single invocation could directly damage the system.

	Chrome			Outlook		
Predicate	$\frac{1}{2}$ Training	All Training	Test	$\frac{1}{2}$ Training	All Training	Test
new-br(*ind*)	113 ± 96	16.1 ± 13.2	5.23 ± 4.27	1k ± 908	537 ± 345	15.4 ± 5.8
new-br(*susp-ind*)	29.4 ± 24.5	4.24 ± 3.49	2.10 ± 1.76	362 ± 272	161 ± 103	5.36 ± 2.22
new-br(*ur*)	1.99 ± 1.93	0.86 ± 0.85	0.02 ± 0.02	153 ± 127	51.2 ± 45.4	0.73 ± 0.65
new-br(*gen-perm*)	0.00 ± 0.00	0.02 ± 0.02	0.15 ± 0.13	2.17 ± 1.91	1.05 ± 1.00	0.64 ± 0.45
new-br(*gen-write*)	0.00 ± 0.00	0.02 ± 0.02	0.01 ± 0.01	1.78 ± 1.51	0.92 ± 0.79	0.36 ± 0.25
to-sys(*new-br*(*ind*), 0x15)	1.00 ± 0.99	2.31 ± 2.26	0.27 ± 0.26	155 ± 145	76 ± 64	0.09 ± 0.09
to-sys(*new-br*(*ind*), 0x25)	0.01 ± 0.01	5.7 ± 5.5	0.23 ± 0.23	1.78 ± 1.67	0.86 ± 0.74	0.09 ± 0.09
to-sys(*new-br*(*ind*), 0x47)	none	0.11 ± 0.10	0.00 ± 0.00	1.89 ± 1.78	0.92 ± 0.78	none
to-sys(*new-br*(*ind*), 0x4d)	none	0.04 ± 0.04	9.8 ± 9.8	2.78 ± 1.34	1.73 ± 0.81	1.00 ± 0.30
to-sys(*new-br*(*ind*), 0x52)	0.02 ± 0.02	0.31 ± 0.30	none	82 ± 73	40.3 ± 37.4	none
to-sys(*new-br*(*ind*), 0xfe)	none	0.01 ± 0.01	none	255 ± 255	124 ± 110	0.55 ± 0.55

	PDF View			PowerPoint View		
Predicate	$\frac{1}{2}$ Training	All Training	Test	$\frac{1}{2}$ Training	All Training	Test
new-br(*ind*)	5.4 ± 5.1	4.19 ± 4.03	0.32 ± 0.31	2k ± 1k	148 ± 92	12.7 ± 5.7
new-br(*susp-ind*)	1.63 ± 1.55	1.28 ± 1.23	0.17 ± 0.17	639 ± 544	37.6 ± 21.7	3.76 ± 1.49
new-br(*ur*)	0.03 ± 0.03	0.02 ± 0.02	0.00 ± 0.00	1.33 ± 1.21	0.20 ± 0.18	0.88 ± 0.88
new-br(*gen-perm*)	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00
new-br(*gen-write*)	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00
to-sys(*new-br*(*ind*), 0x15)	none	0.05 ± 0.03	0.14 ± 0.12	none	none	none
to-sys(*new-br*(*ind*), 0x25)	none	0.05 ± 0.02	0.50 ± 0.25	none	none	none
to-sys(*new-br*(*ind*), 0x47)	none	0.00 ± 0.00	none	none	none	none
to-sys(*new-br*(*ind*), 0x4d)	none	0.02 ± 0.01	0.03 ± 0.03	none	none	none
to-sys(*new-br*(*ind*), 0x52)	none	0.00 ± 0.00	none	none	none	none
to-sys(*new-br*(*ind*), 0xfe)	none	none	none	none	none	none

	SciTE			pdflatex		
Predicate	$\frac{1}{2}$ Training	All Training	Test	$\frac{1}{2}$ Training	All Training	Test
new-br(*ind*)	216 ± 184	184 ± 138	76.4 ± 35.6	24.0 ± 23.5	9.8 ± 9.7	2.93 ± 1.38
new-br(*susp-ind*)	39.8 ± 32.4	32.2 ± 22.9	15.0 ± 4.7	2.90 ± 2.86	1.11 ± 1.10	2.93 ± 1.38
new-br(*ur*)	4.52 ± 3.84	27.2 ± 23.9	2.62 ± 2.25	0.53 ± 0.52	0.24 ± 0.24	0.00 ± 0.00
new-br(*gen-perm*)	0.74 ± 0.66	0.36 ± 0.34	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00
new-br(*gen-write*)	0.22 ± 0.21	0.11 ± 0.11	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00
to-sys(*new-br*(*ind*), 0x15)	0.89 ± 0.82	0.87 ± 0.71	0.75 ± 0.75	13.1 ± 3.8	6.55 ± 2.86	none
to-sys(*new-br*(*ind*), 0x25)	30.2 ± 27.9	15.3 ± 14.3	0.25 ± 0.25	0.29 ± 0.29	0.16 ± 0.15	none
to-sys(*new-br*(*ind*), 0x47)	7.1 ± 6.6	3.67 ± 3.41	0.50 ± 0.50	0.24 ± 0.24	0.13 ± 0.12	none
to-sys(*new-br*(*ind*), 0x4d)	none	0.18 ± 0.17	0.50 ± 0.43	0.44 ± 0.44	0.22 ± 0.21	none
to-sys(*new-br*(*ind*), 0x52)	1.56 ± 1.44	0.93 ± 0.85	1.50 ± 1.36	6.16 ± 2.36	3.11 ± 1.52	none
to-sys(*new-br*(*ind*), 0xfe)	none	none	none	none	none	none

Figure A.2: Predicate convergence for all experiments not reported in Figure 7.1 (lower counts are safer). Label "none" indicates that no instances of the predicate occurred in any execution of the program, while "0.00 ± 0.00" indicates a minuscule quantity (**0x15** NtAllocateVirtualMemory, **0x25** NtMapViewOfSection, **0x47** NtCreateSection, **0x4d** ZwProtectVirtualMemory, **0x52** ZwCreateFile, **0xfe** NtOpenThread).

#	System Call	#	System Call	#	System Call
0x44	NtAddAtom*	0xF0	NtOpenIoCompletion	0x185	NtUnloadKey
0x66	NtAddBootEntry	0xF1	NtOpenJobObject	0x187	NtUnloadKeyEx
0x68	NtAdjustGroupsToken	0xF5	NtOpenKeyedEvent	0x188	NtUnlockFile
0x6C	NtAllocateReserveObject	0xF7	NtOpenObjectAuditAlarm	0x18A	NtVdmControl
0x15	NtAllocateVirtualMemory	0xFB	NtOpenSemaphore	0x18E	NtWaitHighEventPair
0x70	NtAlpcCancelMessage	0xFC	NtOpenSession	0x18F	NtWaitLowEventPair
0x72	NtAlpcCreatePort*	0xFE	NtOpenThread	0x197	NtWow64CsrCaptureMessageBuffer
0x73	NtAlpcCreatePortSection*	0x21	NtOpenThreadToken*	0x191	NtWow64CsrClientConnectToServer
0x74	NtAlpcCreateResourceReserve*	0xFF	NtOpenTimer	0x195	NtWow64CsrFreeCaptureBuffer
0x75	NtAlpcCreateSectionView*	0x100	NtOpenTransaction	0x192	NtWow64CsrIdentifyAlertableThread
0x76	NtAlpcCreateSecurityContext*	0x106	NtPrepareEnlistment	0x19A	NtWow64CsrVerifyRegion
0x77	NtAlpcDeletePortSection*	0x103	NtPrePrepareComplete	0x18	NtWriteFileGather*
0x79	NtAlpcDeleteSectionView*	0x104	NtPrePrepareEnlistment	0x37	NtWriteVirtualMemory*
0x7A	NtAlpcDeleteSecurityContext	0x107	NtPrivilegeCheck	0x60	ZwAcceptConnectPort
0x7B	NtAlpcDisconnectPort	0x109	NtPrivilegedServiceAuditAlarm	0x61	ZwAccessCheck*
0x7D	NtAlpcOpenSenderProcess	0x132	NtRecoverEnlistment	0x26	ZwAccessCheckAndAuditAlarm
0x81	NtAlpcRevokeSecurityContext	0x134	NtRecoverTransactionManager	0x62	ZwAccessCheckByType*
0x49	NtApphelpCacheControl*	0x135	NtRegisterProtocolAddressInformation	0x56	ZwAccessCheckByTypeAndAuditAlarm
0x86	NtCancelIoFileEx*	0x7	NtReleaseSemaphore*	0x63	ZwAccessCheckByTypeResultList
0x5E	NtCancelTimer	0x139	NtRemoveIoCompletionEx	0x64	ZwAccessCheckByTypeResultListAndAuditAlarm
0x38	NtCloseObjectAuditAlarm	0x13C	NtRenameTransactionManager	0x67	ZwAddDriverEntry
0x88	NtCommitComplete	0x13E	NtReplacePartitionUnit	0x3E	ZwAdjustPrivilegesToken
0x8A	NtCommitTransaction	0x9	NtReplyPort*	0x69	ZwAlertResumeThread
0x8B	NtCompactKeys	0x8	NtReplyWaitReceivePort*	0x6B	ZwAllocateLocallyUniqueId
0x8C	NtCompareTokens	0x28	NtReplyWaitReceivePortEx	0x6D	ZwAllocateUserPhysicalPages
0x8E	NtCompressKey	0x13F	NtReplyWaitReplyPort	0x6E	ZwAllocateUuids
0x40	NtContinue*	0x140	NtRequestPort	0x6F	ZwAlpcAcceptConnectPort
0x92	NtCreateEnlistment	0x1F	NtRequestWaitReplyPort	0x71	ZwAlpcConnectPort*
0x45	NtCreateEvent*	0x144	NtResumeProcess	0x78	ZwAlpcDeleteResourceReserve
0x93	NtCreateEventPair	0x145	NtRollbackComplete	0x7C	ZwAlpcImpersonateClientOfPort
0x94	NtCreateIoCompletion	0x149	NtSaveKey	0x7E	ZwAlpcOpenSenderThread
0x96	NtCreateJobSet	0x14B	NtSaveMergedKeys	0x82	ZwAlpcSendWaitReceivePort*
0x98	NtCreateKeyedEvent	0x14D	NtSerializeBoot	0x83	ZwAlpcSetInformation*
0x99	NtCreateMailslotFile	0x14E	NtSetBootEntryOrder	0x84	ZwAreMappedFilesTheSame*
0x9A	NtCreateMutant	0x152	NtSetDefaultHardErrorPort	0x85	ZwAssignProcessToJobObject
0x9C	NtCreatePagingFile	0x154	NtSetDefaultUILanguage	0x5A	ZwCancelIoFile
0xA1	NtCreateProfileEx	0x156	NtSetEaFile	0x87	ZwCancelSynchronousIoFile
0x47	NtCreateSection	0x159	NtSetInformationDebugObject	0x89	ZwCommitEnlistment
0xA6	NtCreateTimer	0x15A	NtSetInformationEnlistment	0x8D	ZwCompleteConnectPort
0xA7	NtCreateToken	0x24	NtSetInformationFile*	0x8F	ZwConnectPort
0xA8	NtCreateTransaction	0x15C	NtSetInformationKey	0x90	ZwCreateDebugObject
0xAA	NtCreateUserProcess	0x59	NtSetInformationObject	0x91	ZwCreateDirectoryObject
0xAB	NtCreateWaitablePort	0x15E	NtSetInformationToken	0x52	ZwCreateFile
0xAD	NtDebugActiveProcess	0x160	NtSetInformationTransactionManager	0x95	ZwCreateJobObject
0xAE	NtDebugContinue	0x161	NtSetInformationWorkerFactory*	0x1A	ZwCreateKey
0x31	NtDelayExecution*	0x162	NtSetIntervalProfile	0x97	ZwCreateKeyTransacted
0xB3	NtDeleteKey	0x165	NtSetLdtEntries	0x9B	ZwCreateNamedPipeFile
0xB4	NtDeleteObjectAuditAlarm	0x166	NtSetLowEventPair	0x9D	ZwCreatePort
0xB5	NtDeletePrivateNamespace	0x168	NtSetQuotaInformationFile	0x9E	ZwCreatePrivateNamespace
0x39	NtDuplicateObject*	0x169	NtSetSecurityObject	0x9F	ZwCreateProcess
0xBA	NtEnableLastKnownGood	0x16A	NtSetSystemEnvironmentValue	0x4A	ZwCreateProcessEx
0xC1	NtFlushInstallUILanguage	0x16C	NtSetSystemInformation	0xA0	ZwCreateProfile
0xC3	NtFlushKey	0x16F	NtSetThreadExecutionState	0xA2	ZwCreateResourceManager
0xC5	NtFlushVirtualMemory	0x5F	NtSetTimer	0xA3	ZwCreateSemaphore
0x1B	NtFreeVirtualMemory	0x170	NtSetTimerEx*	0xA4	ZwCreateSymbolicLinkObject
0xC9	NtFreezeTransactions	0x171	NtSetTimerResolution*	0x4B	ZwCreateThread
0x36	NtFsControlFile*	0x172	NtSetUuidSeed	0xA5	ZwCreateThreadEx
0xD7	NtInitializeRegistry	0x5D	NtSetValueKey	0xA9	ZwCreateTransactionManager
0xD8	NtInitiatePowerAction	0x178	NtStartProfile	0xAC	ZwCreateWorkerFactory
0xDB	NtListenPort	0x17A	NtSuspendProcess	0xAF	ZwDeleteAtom
0xDC	NtLoadDriver	0x17B	NtSuspendThread	0xB0	ZwDeleteBootEntry
0xE2	NtLockRegistryKey	0x17C	NtSystemDebugControl	0xB1	ZwDeleteDriverEntry
0xE6	NtMapCMFModule	0x17D	NtTerminateJobObject	0xB2	ZwDeleteFile
0x25	NtMapViewOfSection	0x29	NtTerminateProcess	0xB6	ZwDeleteValueKey
0xEA	NtNotifyChangeDirectoryFile	0x50	NtTerminateThread	0x4	ZwDeviceIoControlFile*
0xEB	NtNotifyChangeKey	0x180	NtThawTransactions	0xB7	ZwDisableLastKnownGood
0xED	NtNotifyChangeSession	0x5B	NtTraceEvent	0xB8	ZwDisplayString
0x55	NtOpenDirectoryObject	0x182	NtTranslateFilePath	0xB9	ZwDrawText
0x3D	NtOpenEvent	0x183	NtUmsThreadYield	0x3F	ZwDuplicateToken
0x30	NtOpenFile*	0x184	NtUnloadDriver	0xBF	ZwExtendSection

Figure A.3: System calls monitored by DCFI. Numbers correspond to Windows 7 x64 SP 1. Predicate *to-sys*(*new-br*(*ind*), *n*) occurred for those system calls marked with a "*", but these were not reported because they did not occur during the trial, or are not interesting in the combination observed. *(continued on next page)*

#	System Call	#	System Call	#	System Call
0xC0	ZwFilterToken	0x101	ZwOpenTransactionManager	0x15B	ZwSetInformationJobObject
0x48	ZwFlushBuffersFile*	0x102	ZwPlugPlayControl	0x19	ZwSetInformationProcess*
0xC2	ZwFlushInstructionCache	0x5C	ZwPowerInformation	0x15D	ZwSetInformationResourceManager
0xC6	ZwFlushWriteBuffer	0x105	ZwPrepareComplete	0xA	ZwSetInformationThread
0xC7	ZwFreeUserPhysicalPages	0x108	ZwPrivilegeObjectAuditAlarm	0x15F	ZwSetInformationTransaction
0xC8	ZwFreezeRegistry	0x10A	ZwPropagationComplete	0x163	ZwSetIoCompletion*
0xD4	ZwImpersonateAnonymousToken	0x10B	ZwPropagationFailed	0x164	ZwSetIoCompletionEx
0x1C	ZwImpersonateClientOfPort	0x4D	ZwProtectVirtualMemory	0x167	ZwSetLowWaitHighEventPair
0xD5	ZwImpersonateThread	0x10C	ZwPulseEvent	0x16B	ZwSetSystemEnvironmentValueEx
0xD6	ZwInitializeNlsFiles	0x42	ZwQueueApcThread	0x16D	ZwSetSystemPowerState
0xDD	ZwLoadKey	0x12E	ZwQueueApcThreadEx	0x16E	ZwSetSystemTime
0xDE	ZwLoadKey2	0x12F	ZwRaiseException	0x173	ZwSetVolumeInformationFile
0xDF	ZwLoadKeyEx	0x130	ZwRaiseHardError	0x174	ZwShutdownSystem
0xE0	ZwLockFile	0x133	ZwRecoverResourceManager	0x175	ZwShutdownWorkerFactory
0xE1	ZwLockProductActivationKeys	0x136	ZwRegisterThreadTerminatePort	0x176	ZwSignalAndWaitForSingleObject*
0xE3	ZwLockVirtualMemory	0x1D	ZwReleaseMutant	0x177	ZwSinglePhaseReject
0xE4	ZwMakePermanentObject	0x13A	ZwRemoveProcessDebug	0x179	ZwStopProfile
0xE5	ZwMakeTemporaryObject	0x13B	ZwRenameKey	0x17E	ZwTestAlert
0xE7	ZwMapUserPhysicalPages	0x13D	ZwReplaceKey	0x17F	ZwThawRegistry
0xE8	ZwModifyBootEntry	0x141	ZwResetEvent	0x181	ZwTraceControl*
0xE9	ZwModifyDriverEntry	0x142	ZwResetWriteWatch	0x186	ZwUnloadKey2
0xEC	ZwNotifyChangeMultipleKeys	0x143	ZwRestoreKey	0x189	ZwUnlockVirtualMemory
0xEE	ZwOpenEnlistment	0x4F	ZwResumeThread	0x27	ZwUnmapViewOfSection*
0xEF	ZwOpenEventPair	0x146	ZwRollbackEnlistment	0x18B	ZwWaitForDebugEvent
0xF	ZwOpenKey	0x147	ZwRollbackTransaction	0x17	ZwWaitForMultipleObjects32*
0xF3	ZwOpenKeyTransacted	0x148	ZwRollforwardTransactionManager	0x1A3	ZwWow64CallFunction64
0xF4	ZwOpenKeyTransactedEx	0x14A	ZwSaveKeyEx	0x194	ZwWow64CsrAllocateCaptureBuffer
0xF6	ZwOpenMutant	0x14C	ZwSecureConnectPort	0x196	ZwWow64CsrAllocateMessagePointer
0xF8	ZwOpenPrivateNamespace	0x14F	ZwSetBootOptions	0x198	ZwWow64CsrCaptureMessageString
0x23	ZwOpenProcess*	0x150	ZwSetContextThread	0x193	ZwWow64CsrClientCallServer
0xF9	ZwOpenProcessToken	0x151	ZwSetDebugFilterState	0x19B	ZwWow64DebuggerCall
0x2D	ZwOpenProcessTokenEx	0x153	ZwSetDefaultLocale	0x19E	ZwWow64InterlockedPopEntrySList
0xFA	ZwOpenResourceManager	0x155	ZwSetDriverEntryOrder	0x1A1	ZwWow64WriteVirtualMemory64
0x34	ZwOpenSection*	0x2A	ZwSetEventBoostPriority	0x5	ZwWriteFile*
0xFD	ZwOpenSymbolicLinkObject	0x157	ZwSetHighEventPair	0x54	ZwWriteRequestData
0x2C	ZwOpenThreadTokenEx	0x158	ZwSetHighWaitLowEventPair	0x43	ZwYieldExecution

Figure A.4: *(continued)* System calls monitored by DCFI. Numbers correspond to Windows 7 x64 SP 1. Predicate *to-sys*(*new-br*(*ind*), n) occurred for those system calls marked with a "*", but these were not reported because they did not occur during the trial, or are not interesting in the combination observed.