BlackBox: Lightweight Security Monitoring for COTS Binaries

Byron Hawkins and Brian Demsky
University of California, Irvine, USA
{byronh,bdemsky}@uci.edu

Michael B. Taylor
University of California, San Diego, USA
mbtaylor@ucsd.edu

Abstract

After a software system is compromised, it can be difficult to understand what vulnerabilities attackers exploited. Any information residing on that machine cannot be trusted as attackers may have tampered with it to cover their tracks. Moreover, even after an exploit is known, it can be difficult to determine whether it has been used to compromise a given machine. Aviation has long-used black boxes to better understand the causes of accidents, enabling improvements that reduce the likelihood of future accidents.

Many attacks introduce abnormal control flows to compromise systems. In this paper, we present BLACKBOX, a monitoring system for COTS software. Our techniques enable BLACKBOX to efficiently monitor unexpected and potentially harmful control flow in COTS binaries. BLACKBOX constructs dynamic profiles of an application's typical control flows to filter the vast majority of expected control flow behavior, leaving us with a manageable amount of data that can be logged across the network to remote devices.

Modern applications make extensive use of dynamically generated code, some of which varies greatly between executions. We introduce support for code generators that can detect security-sensitive behaviors while allowing BLACKBOX to avoid logging the majority of ordinary behaviors.

We have implemented BLACKBOX in DynamoRIO. We evaluate the runtime overhead of BLACKBOX, and show that it can effectively monitor recent versions of Microsoft Office and Google Chrome. We show that in ROP, COOP, and state-of-the-art JIT injection attacks, BLACKBOX logs the pivotal actions by which the attacker takes control, and can also blacklist those actions to prevent repeated exploits.

Categories and Subject Descriptors D.3.4 [*Software Engineering*]: Processors—security, run-time environments

Keywords Program Monitoring, Control Flow Integrity, Binary Rewriting, Dynamic Code Generation

1. Introduction

Determining how a system was compromised or whether a given exploit was used against a system can be extremely difficult. Information residing on the machine after it is compromised cannot be trusted as attackers may have tampered with or even deleted logs to cover their tracks.

Many exploits leverage software bugs to implement malicious behaviors by manipulating a program's control flow. Indeed, a wide range of control-flow integrity (CFI) tools [13, 23, 24, 28, 33, 36, 37] attempt to block such attacks by constraining program executions to the intended control flow. Unfortunately, it is extremely difficult to exactly determine a program's intended control flow due to fundamental limitations, and many attacks exploit inaccuracy in the knowledge of a program's intended control flow [7, 14, 18, 29, 32].

Rather than attempt to precisely determine a program's intended control flow, we instead focus on developing techniques for (a) efficiently monitoring potential execution anomalies on remote machines to enable later analysis of attacks, (b) distilling and prioritizing logged events to facilitate proactive health monitoring, and (c) blacklisting malicious control flow to prevent known and potential exploits.

Table 1: Distinct modules employing code generators in popular Windows programs. Dynamic code is a growing trend.

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

Dynamically generated code (DGC) poses a significant challenge for monitoring the behavior of applications. Table 1 shows that many major Windows programs incorporate one or more just-in-time (JIT) compilation engines to (a) support internal components developed in scripting languages, or (b) generate small routines that efficiently bind application

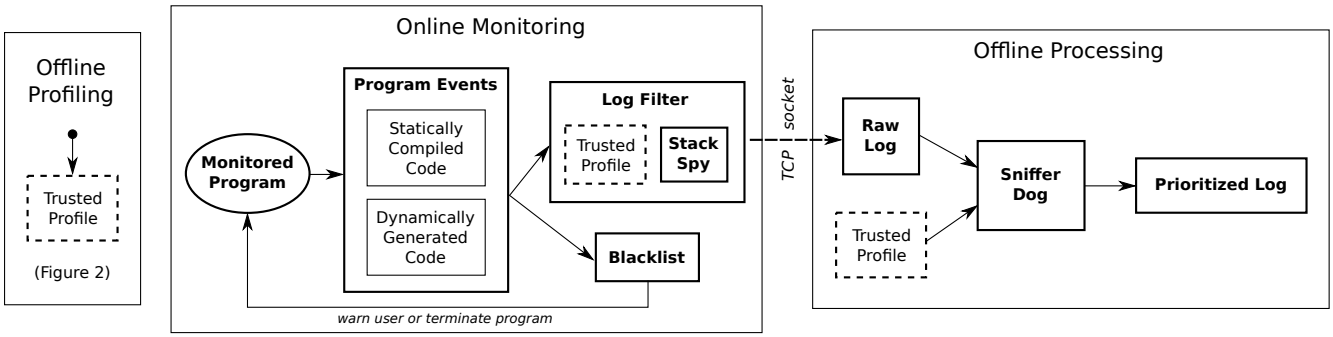

Figure 1: The three phases of BLACKBOX program monitoring.

components at runtime. To avoid logging overwhelming quantities of meaningless control flow events for DGC, it is necessary to learn the behaviors of code generation engines.

1.1 Basic Approach

This paper presents a new tool for efficient black box monitoring of commercial off-the-shelf (COTS) binaries. The key challenge is filtering expected program behaviors, pruning the vast majority of uninteresting data to achieve reasonable instrumentation and logging overheads. Our key insight is that by combining a simple static analysis with online learning, we can extract a *trusted profile* of a program constituting an underapproximation of its intended control flow. Our approach leverages information collected by monitoring the executions of the binaries to learn (1) the normal targets of indirect branches, (2) special cases that appear in real world programs, and (3) the behavior of dynamic code generators.

1.2 Intended Uses

We next discuss some potential use cases for BLACKBOX:

- **Post attack audits:** It can be difficult to determine how a machine was compromised. Organizations often have several similarly configured servers and thus it can be important to determine how a given machine was compromised so that the remaining servers can be secured.
- **Determining whether a given exploit occurred:** After an exploit is known, it can be important to determine which machines have been compromised by that exploit. It may not be sufficient to examine the machine, as the attacker may have carefully hidden any evidence.
- **Statistical analysis:** If an organization has multiple similarly configured machines, statistical analysis can provide insights into attacks. For example, if a specific new behavior appears simultaneously on a few machines, it may be worth investigating whether they were all compromised.
- **Exploit Immunization:** A common security strategy is to blacklist known exploits, but today's anti-virus is labor intensive, requiring (a) manual analysis of malicious binaries and (b) development of signatures to uniquely match binaries on disk. The resulting blacklist performs poorly at the client site, and recent advances in malware development (e.g., payload randomization) make evasion relatively easy.

Figure 1 depicts an overview of the BLACKBOX components, beginning on the left with the Offline Profiling phase, which generates the *trusted profile*. In the Online Monitoring phase (at the end user's site), BLACKBOX filters the majority of benign program events, making it possible to log the remaining events securely to remote servers. The Log Filter simply elides any program event that is already known to the *trusted profile*, with the exception of system calls—since malicious program behavior relies on system calls to carry out its destructive effects, BLACKBOX logs any system call that may have been indirectly affected by untrusted program events. The *stack spy* makes this decision based on program events in the stack frames leading to each system call (Section 4.1). To prevent the exploit of known vulnerabilities, BLACKBOX provides a malware blacklist that can be configured to block specific program events–such as the pivotal branch of a known exploit–or systematically protect error-prone code in the monitored program (Section 5). To further improve usability of the BLACKBOX log, the final Offline Processing phase leverages the *trusted profile* in a statistical analysis that "sniffs out" the most suspicious log entries and prioritizes the log accordingly (Section 4.3).

1.3 Contributions

This paper makes the following contributions:

- **Monitoring Infrastructure.** It presents BLACKBOX, a virtualization system that during a training phase monitors executions to learn a *trusted profile* for a program, greatly reducing the amount of data that must be logged.
- **JIT Code.** It presents an inferred permission system that lifts the control flow variations of dynamically generated code to a level of abstraction that is consistent across executions while retaining its security-sensitive behaviors.
- **Support for Non-standard Control Flow.** It presents techniques for discovering and trusting non-standard control flows that are common in modern Windows programs.
- **Preventative Monitoring.** It presents techniques for prioritizing logged events to facilitate proactive analysis of system health and early vulnerability detection.
- **Control Flow Blacklist.** It presents a control flow blacklist that reliably and efficiently blocks known exploits.

2. Monitoring

BLACKBOX uses binary rewriting to efficiently monitor a client program's control flow. Under binary rewriting, the operating system loads the program modules into memory in the normal way, but the code is no longer executed directly from those mapped images. Instead, BLACKBOX copies program instructions into a code cache as the execution encounters them, and execution occurs over the copy. Since BLACKBOX has exclusive control over the contents of the code cache, the observation of a control-flow branch when it is initially linked within the code cache remains valid for the duration of the execution in the majority of cases. This makes it possible for BLACKBOX to observe every branch while achieving near-native performance (see Figure 5).

The naive approach to black-box monitoring would remotely log every control flow branch taken by each executing thread. But the bandwidth consumption would be enormous, and analysis of such massive logs would be a daunting task. To make the logging approach usable, BLACKBOX employs several techniques to filter out as many *program actions* as can be pre-determined to be safe. To illustrate this approach, Table 3 in Section 6.1 shows samples of log sizes as each *noise reduction* technique is applied.

The first *noise reduction* technique leverages the insight that most exploits rely on some deviation from the normal control flow of the target program—at least one branch occurs that would never be taken outside the influence of the exploit. Instead of logging every branch taken, BLACKBOX only logs each distinct branch once during an execution of the monitored program—at the time that branch is linked within the code cache. This approach retains evidence of every distinct *program action* while astronomically reducing the number of log entries.

2.1 Trusted Profile

The remainder of this section discusses further noise reduction techniques for the BLACKBOX log. These techniques all make use of a *trusted profile*, which is a file on disk contained detailed information about the normal, safe behavior of the monitored program. The key idea is that such ordinary execution behavior need never appear in the remote logs.

The *trusted profile* relies on the fact that the targets of direct branches can be determined offline. For each program module, a static analysis determines the single correct target of each direct branch and adds it to the module's *trusted profile*. At runtime, BLACKBOX consults the profile to determine locally whether a given direct branch is normal, and only logs deviations to the remote server.

The set of branches is represented in the *trusted profile* as a control flow graph (CFG) in which a node represents one basic block of instructions, and an edge represents a transfer of control from one basic block to another, such as a `call` or `jmp`. A basic block is defined as a sequence of instructions having one entry and one exit. To uniquely identify each basic

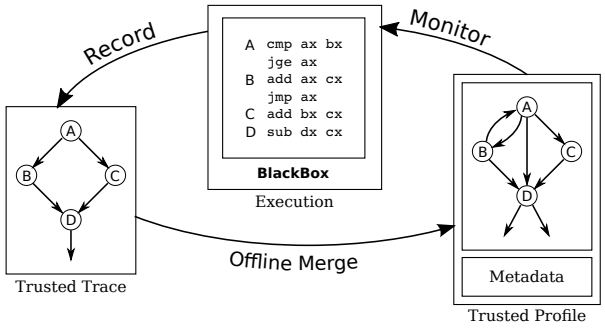

Figure 2: BLACKBOX *trusted profile* training.

block, BLACKBOX labels it with the offset from the start of its containing module. Each edge is labeled with a type, which (for statically compiled code) is one of (a) direct branch, (b) indirect branch, or (c) incorrect return (see Section 2.4). A *program action* is just the execution of a basic block or an edge between them.

2.2 Expanding Trust With Training

To reduce the noise of normal indirect branches in the log, BLACKBOX must learn these branch targets by observation, since indirect branch targets cannot be determined statically without access to source code. For example, the set of functions that may be correctly reached from a call to a C++ instance method can only be determined from a combined analysis of the source code of the program and all its modules. To avoid dependencies on program source code, BLACKBOX employs a training phase in which the program is executed in a trusted environment, and any indirect branch taken by the program is written to a *trusted trace*. An offline tool merges each *trusted trace* into the CFG to extend coverage of the *trusted profile* (as shown in Figure 2).

2.3 Trusting Self-Instrumentation

The *trusted profile* must again be expanded for a special case that commonly arises on the Windows platform: programs often make small changes to their own statically compiled code at runtime. The typical example is system call hooks, in which the program wraps or even replaces a system call by writing a 5-byte `jmp` instruction at the start of the system call trampoline. BLACKBOX must account for two different versions of the same basic block, since these hooks are installed by the program as it starts up. When BLACKBOX observes a hook during the training phase, it expands the unique identifier for the hooked basic block to include the hash code of the basic block instructions, along with a chronological version number.

2.4 Filtering Normal Returns

The most common type of indirect branch—the return—can be systematically filtered from the log in most cases, based on the intuition that if the call was trusted (or has already been logged), the return to the call site must also be trusted (or need not be logged). In a hardware `ret` instruction, a

vulnerable stack entry is consulted to find the location in the calling function to which it should return. To evaluate the correctness (i.e., safety) of call/return pairs, BLACKBOX instruments each call site with a push to a shadow stack, and links all returns to an in-cache assembly routine that pops from the shadow stack and verifies the destination.

Some programs make regular use of *incorrect returns*, for example by deliberately pushing an arbitrary address into the stack slot reserved for the return address and issuing a bogus `ret` instruction. This often occurs in thread pooling mechanisms such as Windows fibers. While these *program actions* are abnormal in the sense of canonical stack usage, BLACKBOX must regard them as normal because they constitute trusted program behavior. When an *incorrect return* occurs during the training phase, BLACKBOX writes it to the *trusted trace* just like any other indirect branch, and handles an *incorrect return* similarly during monitoring.

3. Dynamically-Generated Code

The techniques discussed so far are sufficient for BLACKBOX to efficiently monitor statically compiled Windows programs. But with today's complex development platforms, even a simple program like Windows Notepad leverages the Microsoft Managed Runtime to dynamically generate a few basic blocks in heap-allocated memory. Many low-level security tools give *carte blanche* to DGC, leaving the program vulnerable to arbitrary injection attacks. BLACKBOX cannot apply its *trusted profile* techniques for statically compiled code to DGC because it lacks context information:

- it does not belong to any module on disk,
- the set of possible entry points are only known from the set of observed entry points
- it never has symbols associated with its functions,
- it may use abnormal calling conventions, or omit the call/return convention entirely,
- it may be modified during execution, e.g. the Mozilla Ion JIT frequently toggles opcodes between `lea` and `mov`
- it may be rewritten many times in the same memory location (possibly by code that was dynamically generated),
- a dynamic basic block may rewrite itself as it executes.

To effectively monitor dynamically generated code, BLACKBOX infers a permission model for the program's DGC and writes a concise abstract representation of those permissions to the *trusted profile*. This approach avoids the difficulty of evaluating the DGC itself, and instead trusts any DGC that (a) is produced by the application's trusted code generators, and (b) interacts with the operating system using the application's trusted API for DGC (similar to RockJIT [28], but also applicable to an application's *ad hoc* code generators). When the program's DGC does not comply with the permission model, BLACKBOX only logs the untrusted DGC *program actions* to the remote server. These few deviations are relatively easy for the log analyst to understand, yet provide sufficient information to diagnose exploits, and can even be used to blacklist them.

3.1 Permission Model

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

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

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

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

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

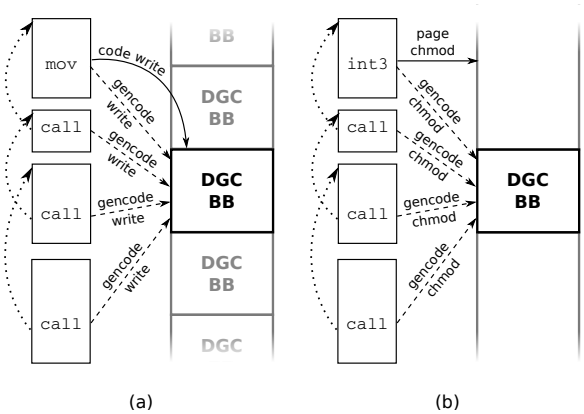

Figure 3: Construction of the (a) *gencode write* and (b) *gencode chmod* edges (dashed arrows) when a code write or page chmod occurs (solid arrows). The `call` sites represent the call chain of the code generator and its dependent libraries (upward dotted arrows).

Figure 3 illustrates the construction of the *gencode write* and *gencode chmod* edges. These edge types are complementary, such that for any DGC basic block, BLACKBOX ob-

serves at least one *gencode write* **or** *gencode chmod*, but not necessarily both. This guarantees visibility of DGC whether it is written directly to an `executable` region, or written to a buffer that is later made `executable`.

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

BLACKBOX observes these three *gencode* action types during the training phase and writes the edges to the *trusted profile*. This becomes the permission model for the program's dynamic code generators. As BLACKBOX monitors the program at the client site, *gencode* actions that conform to these permissions are considered safe, and logging is elided—but if the program takes any other *gencode* actions, the corresponding edges are logged to the remote server. Section 5.1 gives a concrete example of both logging and blacklisting the pivotal attack vectors of a state-of-the-art JIT injection attack.

3.2 Dynamic Singleton Node

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

3.3 Observing Dynamic Code Writes

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

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

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

To mitigate the page fault overhead for store instructions that frequently write code (typical of JIT engines), BLACKBOX can instrument the instruction with a hook to create the *gencode write* edges. This approach greatly improves performance on aggressive JavaScript benchmarks [20].

3.4 Standalone Dynamic Routines

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

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

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

Matching Standalone Dynamic Routines

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

relationship. Candidates having no match are removed from the list, and if the candidate list becomes empty, BLACKBOX marks the standalone as suspicious and logs its current (and any future) basic blocks to the remote server.

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

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

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

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

4. Watchdog Mode

To further improve security on the monitored systems, BLACKBOX can facilitate preventative monitoring with its *watchdog mode*. The key difference between triage and prevention is that triage leverages information about a specific exploit occurrence to narrow its search for the security breach—for example, the paths of affected files—while prevention searches the entire body of program behaviors for potential vulnerabilities, suspicious trends among groups of related users. To this end, *watchdog mode* provides two features that prioritize the logs by estimated degree of suspicion:

- An online *stack spy* that separately logs any system call that occurs in a suspicious stack context, and
- An offline *sniffer dog* that uses PowerLaw modeling to identify log entries having a suspicious smell.

4.1 Stack Spy

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

Table 2: Log entries per day while writing this research paper under BLACKBOX. *Stack spy* highlights any questionable filesystem syscalls (**`0x25`** `NtMapViewOfSection`, **`0x47`** `NtCreateSection`, **`0x52`** `ZwCreateFile`).

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

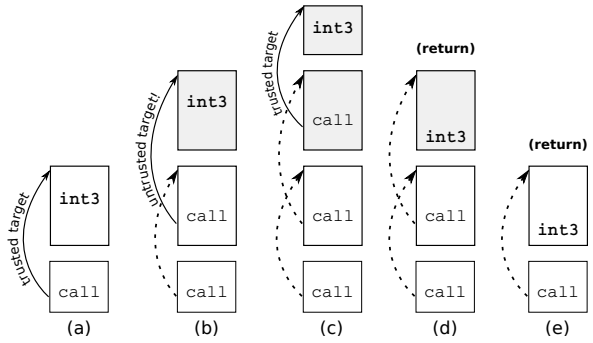

Figure 4: System calls occurring under *stack suspicion* (gray) are logged even if the syscall site is trusted. *Stack spy* raises suspicion in the stack frame where an untrusted program action first occurs (b), and clears suspicion when that stack frame returns (e). The syscalls in (c) and (d) cannot be elided because *stack suspicion* is inherited by callees, but the syscalls in (a) and (e) may be elided because suspicion has not yet been raised (a), or has been cleared (e).

Stack spy implements this feature using a simple boolean flag for each program thread, as illustrated by the function boxes in Figure 4. The flag is initially false (white), and when an untrusted branch occurs (step b), *stack suspicion* is raised at the current stack level (i.e., `esp` in x86 platforms). Any system call made under *stack suspicion* (gray) is logged to the remote server along with the untrusted branch—even if the system call itself is in the *trusted profile* (b, c and d). When the thread eventually returns from the stack frame in which the untrusted branch occurred (e), *stack suspicion* is cleared, and future system calls on that thread can again be trusted.

4.2 Case Study: Authoring Tools

The author of this paper used the SciTE text editor and MikTek pdflatex under *watchdog mode* while writing this research paper and developing the *gencode write* and *gencode chmod* features of BLACKBOX. The *trusted profile* for each

program was trained during the first half of the experiment, and remote logs were accumulated during the second half, as shown in Table 2. More than 100 untrusted indirect branches are logged per day, which represents a significant workload for preventative analysis. But the rate of *suspicious system calls* is fewer than 5 per day, making it possible to efficiently verify that the file system activity from these two programs is not under the influence of malware.

4.3 Sniffer Dog

In addition to spying out the most suspicious system calls, *watchdog mode* provides an offline *sniffer dog* that sorts the most suspicious smelling *program actions* to the top of the log. *Sniffer dog* employs a principle of "typical irregularities" to estimate the probability that an untrusted *program action* is a safe variant of trusted behavior. The iterative process of *trusted profile* training reveals how frequently new edges are normally discovered in each region of the CFG. For example, during profiling of Google Chrome, new indirect branches (and branch targets) are routinely discovered within `chrome_child.dll`—even during the final iterations—since it is a very large module providing a diverse set of features. In contrast, profiling of IISExpress on both static HTML and WordPress (PHP) rarely encounters new edges in the main module `iisexpress.exe`, since its role is limited to server startup and simple routing of requests.

Listing 1: BLACKBOX log for typical usage of Adrenalin.

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

Listing 2: BLACKBOX log of Adrenalin handling a format variation not encountered during *trusted profile* training.

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

To concisely capture these observations, BLACKBOX records a history of new edge discovery between each possible pairing of modules (reflexive included), and summarizes each with a PowerLaw model [2]. *Sniffer dog* consults these models while sorting the log to determine which entries most contradict the typical behavior of the program; log entries conforming to the PowerLaw models are given lower priority, while those exceeding the model's prediction for new events are given higher priority. Listing 1 shows a sorted log for the Adrenalin Media Player in typical usage, Listing 2 shows unusual program activity, and Listing 3 shows an ROP exploit against Adrenalin that launches `calc.exe`.

Listing 3: BLACKBOX log of Adrenalin during an exploit.

```
999 Suspicious entry into DGC
    adrenalinx.dll(0x16f313)→Lib(0x3bfffb) raised suspicion
999 Incorrect return adrenalinx.dll(0x16f313)→Lib(0x3bfffb)
999 Untrusted module calc.exe−ldb1446a00060001
999 Suspicious indirect shlwapi.dll(0x1c508)→Lib(0x192aa7)
999 Suspicious indirect ntdll.dll(0x3c04d)→Lib(0xe2e831)
999 Untrusted module gdiplus.dll−ldb146c800060001
999 Suspicious indirect kernel32.dll(0x13365)→Lib(0xdb3fc3)
998 DGC standalone owned by adrenalinx.dll-300010001 (4 nodes)
900 Suspicious syscall #25 ZwSetInformationProcess
    ntdll.dll(0x22373→0x224b0) raised suspicion
900 Suspicious syscall #82 ZwCreateFile
    ntdll.dll(0x2239c→0x22468) raised suspicion
900 Suspicious syscall #79 ZwResumeThread
    kernelbase.dll(0x14148)→Lib(0x6ee3a) raised suspicion
900 Suspicious syscall #26 ZwCreateKey
    user32.dll(0x16d88)→Lib(0xd5f105) raised suspicion
900 Suspicious syscall #77 ZwProtectVirtualMemory
    apphelp.dll(0x13066)→Lib(0xd5f105) raised suspicion
900 Suspicious syscall #165 ZwCreateThreadEx
    kernelbase.dll(0x13f6d)→Lib(0xa57647) raised suspicion
... (flood of similar entries)
```

As an offline tool, *sniffer dog* is also able to generalize observations about log entries from groups of monitored systems. For example, if many users within a certain organization or business unit encounter a particular untrusted *program action* at a significantly higher rate than others, *sniffer dog* may raise this anomaly as a security smell—it may indicate a vulnerability or program error, or just that the group is downloading a questionable third-party plugin. Standalone third-party products can also interfere with system health, if for example they install drivers or shared libraries. Early discovery of these symptoms makes it possible to identify and correct the issue before any inadvertent damage occurs on monitored systems. In cases where it becomes necessary to file a bug report with a software vendor, the detailed BLACKBOX logs will greatly simplify the debugging effort, leading to faster and more accurate patches.

5. Malware Blacklist

A key complement to the preventative features of *watchdog mode* is the ability to explicitly blacklist known exploits. This functionality is traditionally provided by anti-virus, which relies on large-scale manual labor by highly skilled experts. First, *malware diagnosis* determines the symptoms of an infected binary by executing it in an analysis sandbox. Malware is often designed to probe for such a sandbox and hide from the analysis. Once the effects of a virus are understood, a *binary signature* is generated to uniquely match infected files on disk. Randomized payloads dramatically increase the workload of generating binary signatures, since each variant of the same malware may require a completely

different signature. As malware development tools increase in sophistication and efficiency, the cost of the traditional anti-virus approach rises, even as its reliability is eroded.

The BLACKBOX monitoring system provides a low-cost alternative to *malware diagnosis* that is immune to these evasion tactics. Malware has nothing to gain by withholding its payload because BLACKBOX monitors programs at the end user's site. And superficial randomization will not help malware evade detailed, low-level observation.

BLACKBOX also provides a low-cost, reliable alternative to *binary signature matching*. Instead of attempting to uniquely identify a malware instance by its representation on disk, BLACKBOX can be configured with a control flow blacklist that identifies an exploit at the specific point it compromises the monitored program. This approach is similarly immune to superficial randomization because malware has a very limited number of opportunities to exploit a given program. A blacklist entry specifies:

1. The module-relative address of the basic block to protect,
2. The *program action* to prohibit, and
3. An alternative action to take, such as:
   a. Stop with an error explaining the exploit, including details about the form and source of the crafted input.
   b. Redirect execution to a global error handler (may require collaboration with the program vendor).

5.1 Case Study: Blocking Code Injections

Recent advances in browser security make it much more difficult for an attacker to gain control of compiled JavaScript. For example, the Chakra JIT engine in Microsoft Internet Explorer (IE) places all JavaScript data in non-`executable` pages, and obfuscates any JavaScript constant larger than 2 bytes. While these techniques make it increasingly difficult to exploit the browser via JavaScript, [4] demonstrates working examples of a code injection that leverages return-oriented programming (ROP) to compromise a recent version of IE.

BLACKBOX will log several untrusted *program actions* during this attack, which takes the following sequence:

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

To train the *trusted profile* of the Chakra JIT engine, the author of this paper used Microsoft Outlook for email during a 4-week period. The profile contains no edges from the Chakra *dynamic singleton* to system calls, and no self-edges of type *incorrect return* or *gencode chmod*. Suppose he now receives an email containing the crafted JavaScript:

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

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

5.1.1 Automated Blacklisting

The BLACKBOX *sniffer dog* may be able to automatically detect potential exploits and blacklist them proactively. For example, in our experience, any given `ret` instruction is used exclusively for either normal or incorrect returns—throughout our extensive corpus of execution logs, which includes hundreds of usage hours for large Windows programs, no `ret` instruction is ever used for a normal return at one time, and for an *incorrect return* at another time. This observation makes it possible for *sniffer dog* to simply collect all instances of normal returns throughout the monitored programs and blacklist the *incorrect return* action at each one. While it may still be possible for an adversary to compromise a `ret` instruction that lies outside the *trusted profile* for a given program, it greatly reduces both the number of exploit opportunities, and the number of end users who will be affected by a successful exploit.

5.2 Case Study: Blocking COOP

Counterfeit Object-Oriented Programming [32] is designed to thwart control-flow integrity (CFI) schemes: by injecting the target program with bogus objects having crafted virtual dispatch tables, this exploit deviously conforms to category-based CFI policies that only constrain the protected program to make method calls at method call sites. The BLACKBOX *trusted profile*, however, does not contain phony branch targets, so the hijacked calls in a COOP attack will be logged to the remote server as they occur.

While it would be ideal to blacklist all indirect branch targets not appearing in the *trusted profile*, this is not generally possible—our experiments show that in large Windows programs, normal new branch targets do occasionally occur for known indirect branches. But for a given COOP attack,

the BLACKBOX *stack spy* can isolate branches leading to the system calls that comprise the payload, making it relatively easy for the log analyst to blacklist the pivotal attack points.

6. End-to-End Results

To complement the case studies in Sections 4 and 5, we present a quantitative evaluation of BLACKBOX[1] in (a) real world usage scenarios, (b) exploit detection, and (c) performance benchmarks. We conduct the experiments in Windows 7 SP 1 running in VirtualBox 4.2.10 on an Ubuntu 13.04 host using an Intel Xeon E3-1245 v3 CPU. The Windows Update service and application updates are disabled to maintain consistency throughout the experiments.

6.1 Filtering Log Noise

For BLACKBOX to be usable in real deployment scenarios, its *noise reduction* techniques must be effective across a broad range of popular programs, content types, runtime environments and user interface habits. In this set of experiments, we trained the *trusted profile* of each program with a workload that could be scripted to execute on a cluster of Windows desktop machines. For programs offering a rich user interface, we enumerated the input options (such as ribbon menu, right-click menu, shortcut key, etc.) and exercised them in combination until the discovery of new CFG elements in the *trusted profile* converged to a nominal rate. We accounted for the various domains of program flexibility in a similar way (for example, file storage to local disk, LAN/WAN, cloud storage, etc.), resulting in a *trusted profile* having dense code coverage within the set of profiled features.

After deploying BLACKBOX with the *trusted profile*, we manually used the program for a period of several hours to several days, repeating a portion of the original training material, along with additional input content that had been deliberately withheld from profiling. Table 3 shows the *noise reduction* of each successive BLACKBOX technique from left to right. After the final learning phase ("Learning Indirects"), the logs focus concisely on the security sensitive *program actions*. Table 4 presents a similar result for the special cases of *incorrect return* and the DGC edges.

6.2 Logging and Blacklisting Exploits

Three published exploits were executed against programs monitored by BLACKBOX to verify that (a) each *program action* induced by the exploit is logged to the remote server, and (b) control flow stops at blacklisted *program actions*. It was not possible to obtain working exploits for the more popular Windows programs, due to bounties paid by vendors to prevent distribution or replication of those attacks. The following training procedure for the *trusted profile* was shared with the previous experiment on *noise reduction*:

[1] The BLACKBOX implementation is open source and can be found at `http://www.github.com/uci-plrg/blackbox`.

Table 4: Average number of log entries before and after the learning phase of the *trusted profile* for an hour of normal program activity (lower is better).

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

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

Upon deploying the *trusted profile* in each experiment, we also blacklisted the exploit at various attack points—the *incorrect return*, an untrusted indirect branch, and a *suspicious system call*—and configured the blacklist to pause with a dismissable warning. The warning did not interrupt normal usage of the programs, but promptly appeared when we executed the exploit. As expected, BLACKBOX logged the *incorrect return* followed by a sequence of untrusted indirect branches and *suspicious system calls* that forked the payload (see Listing 3 in Section 4.3 for the Adrenalin log).

6.3 IIS

To demonstrate the effectiveness of BLACKBOX in server applications, we trained the *trusted profile* of IIS Express 7.0 and PHP 5.4.28 on typical web content: a 2GB snapshot of cbssports.com, the PHP unit test suite, and a default install of WordPress. Before deploying this *trusted profile*, we performed several WordPress upgrades: installing the popular e-commerce plugin WooCommerce, activating the anti-spam plugin Akismet, changing the theme, upgrading the WordPress core, and importing posts into the blog and

Table 3: Average number of log entries during an hour of normal program activity. Each column shows the log size as each of the BLACKBOX *noise reduction* techniques is progressively applied from left to right (lower is better).

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

products into the store. Then we ran a form-enabled crawler on the upgraded WordPress site for 6 hours, sending a minimum of 20,000 unique requests to the major areas of the site: (1) store configuration and product editing, (2) blog administration and post editing, (3) theme customization, (4) public pages, and (5) upload forms. Despite having added major functionality outside the scope of the *trusted profile*, BLACKBOX only logs 39 indirect branches (Table 5).

Table 5: Total log entries during 6 hours of fuzz testing WordPress on IIS (lower is better). The *trusted profile* was trained on a default WordPress installation, but fuzz testing was executed after two plugins were installed, the theme was changed, and the WordPress core was updated.

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

6.4 Performance

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

To more accurately understand the execution overhead, we employ industry standard benchmarks that focus on compute performance. We evaluated the performance of BLACKBOX relative to native execution on the SPEC CPU 2006 benchmark suite [21], which consists of a diverse set of CPU bound applications across several application domains and languages: 7 C++ programs, 12 C programs, 4 C/Fortran programs, and 6 Fortran programs. Across the suite of benchmarks we measured a geometric mean of 14.7% slowdown; Figure 5 presents the individual overheads. Benchmarks having mostly direct branches incur minimal (or zero) overhead in BLACKBOX, while programs having a greater proportion of indirect branches with multiple targets (typically C++ programs and script interpreters) incur the higher overheads. Profile-guided optimizations for high-degree indirect branches show promising results, but we reserve the formal evaluation for future work. While some of the larger Windows programs in our experiments do have a high rate of indirect branches, the majority are cross-module function calls through the IAT that by construction have only one target, making it feasible for BLACKBOX to optimize them as direct branches. Earlier work addresses the performance of DynamoRIO on dynamically generated code [20].

7. Related Work

BLACKBOX logs security-sensitive control flow activity in deployed systems, whereas TraceBack [5] focuses on debugging support, and Schnauzer [6] profiles the control flow frontier for potential security vulnerabilities.

BLACKBOX is distinguished from CFI techniques by its dynamic approach to discovering the target program's normal control flow. The key advantages of BLACKBOX relative to other approaches [35] are that it can handle the special cases that arise in real world code, self-modifying code, dynamic code generation, and can learn more precise control flow information as it is not constrained by the limitations of static analysis. It can monitor and blacklist events from attacks that would slip past other CFI approaches.

7.1 Automated Whole-Program CFI Defenses

Automated CFI defense is an all-or-nothing proposition: it can potentially block attacks while adding minimal runtime and administrative overhead—but if it fails to block an attack, the defense offers no other benefit. For example, Context-Sensitive CFI [34] reports overhead under 10% (for certain applications) and is capable of defeating advanced attacks such as Control Flow Bending [8] (CFB) and Control Flow Jujutsu [17], but it is not invincible to every instance of these

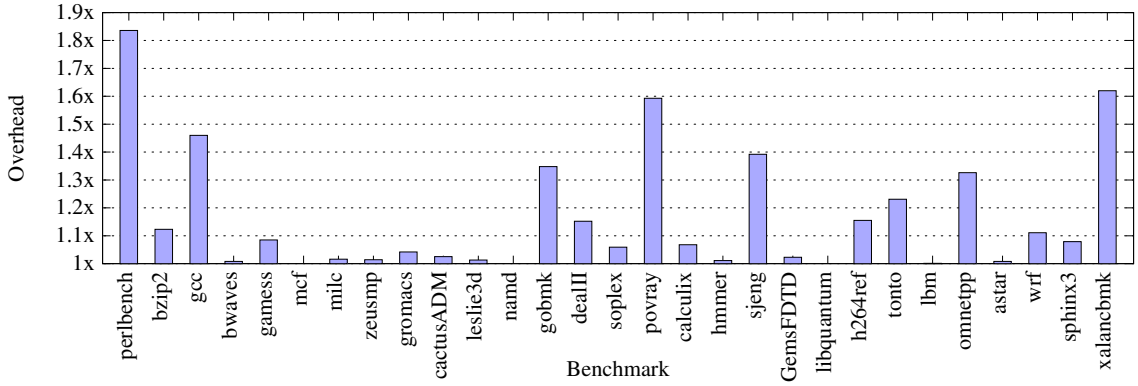

Figure 5: Normalized BLACKBOX execution times for Spec CPU 2006, taking the geometric mean of 3 runs (lower is better).

attacks. Cryptographic CFI [26] has overhead under 20% (for many programs) and can defeat CFB and Jujutsu by securing forward and backward edges with a MAC, but cannot always defeat COOP and fails to protect the kernel-managed return address of the `sysenter` instruction that is targeted by StackDefiler [11]. Coarse-grained CFI approaches such as CCFIR [36] and binCFI [37] report overhead under 10% on static benchmarks, but can be defeated by gadget synthesis attacks [14, 18], along with these more advanced attacks.

We are not aware of any control flow attack that can evade or disable the BLACKBOX monitor. While the blacklist offers the most effective protection, it is not the only line of defense—untrusted *program actions* will always be logged to a secure server. This reliability comes at a cost of administrative overhead: training the *trusted profile*, configuring the blacklist, and monitoring the BLACKBOX logs. Runtime overhead is moderate for most programs (14.5% for SPEC CPU 2006), though it can approach 2× for programs that heavily rely on high-degree indirect branches.

7.2 Other CFI Techniques

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. Tice *et al*. [33] developed a similar compiler-based approach that protects only forward control flow edges. Control-Data Isolation [3] rewrites both forward and backward edges with exclusively direct branches, but requires recompilation of all modules. These approaches have overhead under 10% and can defeat COOP and some other advanced attacks, but do not protect DGC and require source code to be available.

OpaqueCFI [27] uses randomization combined with simple range checks to implement CFI. It is largely orthogonal to our work—it does not attempt to improve the precision of CFI, nor does it seek to solve the problems associated with implementing CFI for COTS modern Windows applications. Abadi *et al*. proposed an early approach to CFI [1] that targeted Windows binaries. Their approach cannot handle dynamic code or binaries that violate compiler conventions.

Several approaches target specific attacks [9, 10, 12, 29, 30], such as ROP, but may not be robust to new types of attacks. Recent work has shown that many of these approaches are vulnerable to modified versions of the attacks [7, 19].

Clearview [31] 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 to restore the invariants. While BLACKBOX 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 BLACKBOX focuses on detecting attacks.

XFI is a static rewriter-based approach to CFI [15]. It checks coarser grained constraints on control flow than BLACKBOX, 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 [13]. It uses static analysis to extract the CFG and requires statically unresolvable indirect jumps target a function entrance. Program shepherding enforces various execution policies such as code origins on program executions [23], but enforces weaker properties than CFI systems.

RockJIT [28] integrates CFI support into a JIT, protecting the code generation process and constraining the generated code to its intended API. Our approach is comparable, yet also compatible with off-the-shelf JIT engines.

Code Pointer Integrity places pointers to code in a separate protected heap [24], though [16] can defeat this approach.

A collaborative approach [25] lowers the overhead of CFI by distributing the checking workload across users of an application until an attack is detected, at which point all members of the application community check for the attack.

8. Conclusion

We presented BLACKBOX, a tool for monitoring control flow based on previous observations of a program at runtime. BLACKBOX handles dynamic code generation by inferring a permission system on code generators. BLACKBOX can diagnose and blacklist exploits. Our experience indicates that BLACKBOX can effectively monitor real world applications and is able to filter an execution's control flow sufficiently to capture novel behaviors that arise in attacks while keeping the total volume of data manageable for remote logging.

Acknowledgments

We thank Peizhao Ou, Bin Xu, Rahmadi Trimananda, Per Larsen, Andrei Homescu and the anonymous reviewers for their helpful comments. This work was supported by the National Science Foundation under award 1228992, grants CCF-0846195, CCF-1217854, CNS-1228995, and CCF-1319786; and by the Center for Future Architectures Research (C-FAR), one of six centers of STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA.

References

[1] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity. In *CCS*, 2005.

[2] J. Alstott, E. Bullmore, and D. Plenz. powerlaw: A python package for analysis of heavy-tailed distributions. *PLoS ONE*, 9(1):e85777, 2014. doi: 10.1371/journal.pone.0085777.

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

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

[5] A. Ayers, R. Schooler, C. Metcalf, A. Agarwal, J. Rhee, and E. Witchel. Traceback: First fault diagnosis by reconstruction of distributed control flow. In *PLDI*, 2005.

[6] V. Bertacco, R. Rodriguez, W. Arthur, B. Mammo, and T. Austin. Schnauzer: Scalable profiling for likely security bug sites. In *CGO*, 2013.

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

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

[9] P. Chen, H. Xiao, X. Shen, X. Yin, B. Mao, and L. Xie. DROP: Detecting return-oriented programming malicious code. In *ICISS*, 2009.

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

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

[12] L. Davi, A.-R. Sadeghi, and M. Winandy. ROPdefender: A detection tool to defend against return-oriented programming attacks. In *ASIACCS*, 2011.

[13] 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 *NDSS*, 2012.

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

[15] U. Erlingsson, S. Valley, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula. XFI: Software guards for system address spaces. In *OSDI*, 2006.

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

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

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

[19] E. Göktas, E. Athanasopoulos, M. Polychroniakis, H. Bos, and G. Portokalidis. Size does matter - why using gadget chain length to prevent code-reuse attacks is hard. In *USENIX Security*, 2014.

[20] B. Hawkins, B. Demsky, D. Bruening, and Q. Zhao. Optimizing binary translation for dynamically generated code. In *CGO*, 2015.

[21] J. L. Henning. SPEC CPU2006 benchmark descriptions. *SIGARCH Computer Architecture News*, 2006.

[22] D. Jang, Z. Tatlock, and S. Lerner. SafeDispatch: Securing C++ virtual calls from memory corruption attacks. In *NDSS*, 2014.

[23] V. Kiriansky, D. Bruening, and S. Amarasinghe. Secure execution via program shepherding. In *USENIX Security*, 2002.

[24] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song. Code-pointer integrity. In *OSDI*, 2014.

[25] M. E. Locasto, S. Sidiroglou, and A. D. Keromytis. Software self-healing using collaborative application communities. In *NDSS*, 2005.

[26] A. J. Mashtizadeh, A. Bittau, D. Boneh, and D. Mazières. Ccfi: Cryptographically enforced control flow integrity. In *CCS*, 2015.

[27] V. Mohan, P. Larsen, S. Brunthaler, K. W. Hamlen, and M. Franz. Opaque control-flow integrity. In *NDSS*, 2015.

[28] B. Niu and G. Tan. Rockjit: Securing just-in-time compilation using modular control-flow integrity. In *CCS*, 2014.

[29] V. Pappas, M. Polychronakis, and A. D. Keromytis. Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. In *S&P*, 2012.

[30] V. Pappas, M. Polychronakis, and A. D. Keromytis. Transparent ROP exploit mitigation using indirect branch tracing. In *SEC*, 2013.

[31] J. H. Perkins, S. Kim, S. Larsen, S. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, S. Sidiroglou, G. Sullivan, W.-F. Wong, Y. Zibin, M. D. Ernst, and M. Rinard. Automatically patching errors in deployed software. In *SOSP*, 2009.

[32] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A.-R. Sadeghi, and T. Holz. Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in C++ applications. In *S&P*, 2015.

[33] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, Úlfar Erlingsson, L. Lozano, and G. Pike. Enforcing forward-edge control-flow integrity in GCC & LLVM. In *USENIX Security*, 2014.

[34] V. van der Veen, D. Andriesse, E. Göktaş, B. Gras, L. Sambuc, A. Slowinska, H. Bos, and C. Giuffrida. Practical context-sensitive CFI. In *CCS*, 2015.

[35] Y. Xia, Y. Liu, H. Chen, and B. Zang. CFIMon: Detecting violation of control flow integrity using performance counters. In *DSN*, 2012.

[36] 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 *S&P*, 2013.

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