JITScope: Protecting Web Users from Control-Flow Hijacking Attacks

Chao Zhang*, Mehrdad Niknami*, Kevin Zhijie Chen*, Chengyu Song†, Zhaofeng Chen‡, Dawn Song*

*University of California, Berkeley
{chaoz, mniknami, kevinchn, dawnsong}@cs.berkeley.edu

‡Peking University
chenzhaofeng@pku.edu.cn

†Georgia Institute of Technology
csong84@gatech.edu

***Abstract*—Web browsers are one of the most important end-user applications to browse, retrieve, and present Internet resources. Malicious or compromised resources may endanger Web users by hijacking web browsers to execute arbitrary malicious code in the victims' systems. Unfortunately, the widely-adopted Just-In-Time compilation (JIT) optimization technique, which compiles source code to native code at runtime, significantly increases this risk. By exploiting JIT compiled code, attackers can bypass all currently deployed defenses.**

In this paper, we systematically investigate threats against JIT compiled code, and the challenges of protecting JIT compiled code. We propose a general defense solution, `JITScope`, to enforce Control-Flow Integrity (CFI) on both statically compiled and JIT compiled code. Our solution furthermore enforces the W⊕X policy on JIT compiled code, preventing the JIT compiled code from being overwritten by attackers. We show that our prototype implementation of `JITScope` on the popular Firefox web browser introduces a reasonably low performance overhead, while defeating existing real-world control flow hijacking attacks.

I. Introduction

Web browsers are one of the most important end-user applications. PC and smartphone users use browsers to browse, retrieve and present Internet resources. However, the browser may receive malicious resources while browsing the internet when the remote server is compromised, or when the connection to the server is tampered with (e.g. by performing man-in-the-middle attacks [1]). These malicious resources may launch application-level attacks such as cross-site scripting (XSS [2]) and cross-site request forgery (CSRF [3]), or they may even launch control-flow hijacking attacks that cause web browsers to execute arbitrary malicious code (such as shellcode) in the victims' systems.

Control-flow hijacking attacks have a long history in software security. Attackers usually break the program state (comprising of the code and data in memory) by exploiting vulnerabilities in target applications, then hijack the control-flow of target applications to execute malicious code. These types of attacks are still the dominant threats to modern systems. The impact of these attacks is more severe for network applications such as browsers, not only because they deal with more untrusted input sources than traditional desktop applications, but also because they are used to perform many sensitive user activities, including online banking, shopping and tax reporting.

Modern operating systems and compilers have deployed several solutions to mitigate control-flow hijacking attacks, including Address Space Layout Randomization (ASLR [4]) and Data Execution Prevention (DEP or W⊕X [5]). Researchers have also proposed solutions that enforce Control Flow Integrity (CFI [6–10]), to defeat control-flow hijacking attacks. But applications such as web browsers use Just-In-Time compilation (JIT) to compile scripts (e.g., JavaScript) to native code at runtime and then execute them, making the traditional boundary between code and data more vague, thus causing most existing defenses ineffective. For example, attackers can use heap-spray attacks [11] or information leakage attacks [12] to bypass ASLR. Furthermore, they can use scripts to write shellcode [13] to bypass W⊕X.

Moreover, the JIT optimization technique makes the defense more challenging. First, because the JIT compiled code is generated at runtime, it resides in memory that is both writable and executable (and thus not protected by W⊕X). Attackers can override any defense instrumented in the writable JIT memory. Second, by predicating the behaviors of JIT compilers and feeding them with specially crafted inputs (i.e., specific scripts), attackers can manipulate the layout of the JIT memory to launch various types of attacks, such as JIT spraying [14–17]. Furthermore, this prevents the application of defenses such as CFI, as it is difficult to obtain the control-flow information for the JIT compiled code.

Researchers have proposed some specific defense solutions to protect the JIT compiled code. For example, the librando [18] and INSeRT [19] solutions randomize the behaviors of JIT compilers to stop JIT spraying attacks. However, they only provide a probabilistic defense. NaCl-JIT [20] and RockJIT [21] enforce coarse-grained CFI on the JIT compiled code with sandbox techniques. Their performance overheads, however, are too high.

In this paper, we propose a general defense solution, `JITScope`, to enforce a strong security policy with a low performance overhead, used to protect applications (such as web browsers) that support JIT compilation from control-flow hijacking attacks. More specifically, we deploy a fine-grained CFI policy on statically compiled code during compilation, and wrap the JIT compiler to deploy a coarse-grained CFI policy on the JIT compiled code. We also apply a W⊕X policy on the JIT memory, to enforce the integrity of the JIT compiled

978-1-4799-8381-0/15/$31.00 ©2015 IEEE

code and the CFI instrumentation.

Additionally, to ensure a lower performance overhead, we only use CFI to protect forward edges (i.e., indirect call and jump instructions), and utilize the shadow stack solution to protect backward edges (i.e., return instructions). The shadow stack also provides an accurate target match for return instructions, providing a stronger protection than CFI.

We implement a prototype of `JITScope` based on the LLVM compiler infrastructure [22], and apply it on the web browser Firefox and its latest JIT compiler IonMonkey. To the best of our knowledge, this is the first complete defense solution applied not only to the JIT compiler, but also to the full application. Our results show that `JITScope` introduces a reasonable performance overhead (less than 10%) lower than all existing solutions, while demonstrating that hardening Firefox by `JITScope` allows it to defeat existing real-world attacks.

In summary, this paper makes the following contributions:

- We summarize the security risks of web browsers, especially threats against the JIT compiled code, and point out the challenges of protecting JIT compilers.
- We describe the primitives of a practical defense against threats to web browsers, and we propose a general defense solution `JITScope` to protect web browsers from control-flow hijacking attacks.
- We implement a prototype of `JITScope` and apply it to a full web browser, including its JIT compiler. Results show that this solution is efficient and effective. The performance overhead is less than 10%, which is lower than existing solutions, and the hardened application can defeat existing real-world control-flow hijacking attacks.

The remainder of this paper is organized as follows: We discuss related work in Section II and the problem definition in Section III. Sections IV and V describe the design and implementation of `JITScope`. Section VI gives the evaluation result of `JITScope`, and finally Section VII concludes our discussion.

II. Related Work

In this section, we talk about related work on general control-flow hijacking attacks and defenses first, and then discuss the related work on JIT-related attacks and defenses.

A. Control-Flow Hijacking Attacks and Defenses

Researchers have proposed many defense solutions to defeat control-flow hijacking attacks. Operating systems and compilers have deployed some of them in practice, including StackGuard [23] that detects return address tampering, W$\oplus$X [5] that prevents memory from being both executable and writable, and ASLR [4] that randomizes code location in memory.

Attackers can, however, bypass all these defenses. They can exploit other control data such as function pointers to bypass StackGuard and launch various attacks, such as vtable hijacking attacks [24, 25]. Attackers can reuse existing code snippets to stitch up a shellcode and bypass W$\oplus$X, as is the case with the return-to-libc [26] or ROP [27] attacks. Attackers can also exploit certain vulnerabilities to read arbitrary or specific memory and launch information leakage attacks [28], bypassing the secret-based solutions such as ASLR and StackGuard.

Researchers have proposed several new defense solutions to defeat these advanced attacks.

1) Control-Flow Integrity: The CFI solution provides a strong guarantee that all control flow transfers must comply with programmers' intentions, i.e., they must respect the program's compile-time Control-Flow Graph (CFG). This can stop various types of control flow hijacking attacks, including ROP, return-to-libc, and vtable hijacking attacks. The original CFI solution was proposed in 2005 [6], but it has not been adopted by the industry due to several limitations.

Recently proposed coarse-grained CFI solutions [7, 8] deploy CFI directly on binary executables. As there is no type information available in binaries, however, only a coarse-grained CFI policy is enforced, and attackers can therefore bypass these protections in some cases [29].

Other CFI solutions [9, 10] enforce a fine-grained CFI policy on target applications. These solutions utilize the information collected by compilers at compile-time or by virtual machines. They usually provide a stronger protection than coarse-grained CFI solutions. However, the higher performance overhead also restricts their adoptions. A recent solution [30] enforces the CFI policy on only forward-edges (i.e., indirect call and jump instructions) on Chrome, and obtains an acceptable performance overhead (about 4%).

The NaCl-JIT [20] solution is the first CFI solution deployed on JIT compiled code. It uses the Software Fault Isolation (SFI) [31] based sandbox technique to enforce that all indirect control transfer in JIT compiled code only jump to aligned code address. This is also a coarse-grained policy, even weaker than the aforementioned solutions [9, 10]. Moreover, it introduces a high performance overhead (about 50%) prohibiting its adoption. The RockJIT [21] solution also combines coarse-grained CFI and sandbox to protect JIT compiled code, incurring a performance overhead of about 15%.

2) Memory Safety: The memory safety policy ensures that no out-of-bounds or dangling pointers can be exploited to get unauthorized access to memory. As a result, attackers cannot tamper target applications' states, or launch control-flow hijacking attacks. Researchers proposed many memory safety solutions [32–35]. The *spatial* memory safety solution SoftBound [36] and the *temporal* memory safety solution CETS [37] are two representative solutions.

These solutions usually instrument pointers with extra metadata when they are created, track the metadata during the program execution, and then check the metadata when these pointers are used to access memory. These solutions, however, all introduce a high performance overhead, prohibiting their adoptions. For example, the combination of SoftBound and CETS will enforce *complete* memory safety at the cost of $2\times$ or more performance overhead.

Code Pointer Integrity (CPI [38]) proposed a lightweight memory safety solution, protecting only sensitive pointers including code pointers, capable of defeating control-flow hijacking attacks for statically compiled code. However, it does not provide protections for JIT compiled code.

B. JIT-related Attacks and Defenses

The widely adopted optimization technique JIT compilation also brings security risks to users.

1) JIT Code Corruption and Injection: The JIT compiled code memory is both executable and writable, making the classic W⊕X defense inapplicable. Attackers can exploit some vulnerabilities and overwrite the JIT code memory to corrupt the JIT code or inject malicious code, and then to divert the control flow to this corrupted or injected code.

The NaCl-JIT [20] and RockJIT [21] solutions can mitigate this type of attacks by sandboxing all memory write operations to eliminate unauthorized write operations. The solution [39] enforces the JIT memory to be non-writable, and delegates all JIT memory write operations to a trusted process that shares the JIT memory with the browser. Our solution `JITScope` can also defeat these type of attacks by wrapping the JIT compiler and enforcing the W⊕X policy.

2) JIT Code Reuse: Attackers may manipulate the JIT code memory layout by feeding the JIT compiler with specific inputs (e.g., scripts), and then reuse existing JIT code to launch attacks, e.g., JIT spraying. After the expected memory layout is deployed, attackers may divert the control flow to a specific address in this controlled memory, e.g., jump to the middle of an instruction and launch the classic ROP attacks.

Software diversity solutions, e.g., librando [18], INSeRT [19] and JITSafe [40, 41], randomize the behavior of JIT compilers, and thus randomize the generated JIT code to stop this type of attack. These solutions usually change the code generation logic of the JIT compilers, by inserting padding bytes, replacing instructions with equivalent ones, and etc.

The NaCl-JIT [20] and RockJIT [21] solutions can block illegal control flow. Thus, even if attackers successfully deploy the JIT code memory, they cannot divert the control flow to this memory region. These solutions, however, incur a high performance overhead. Our solution `JITScope` defeats this type of attack by enforcing CFI with a reasonable overhead.

III. Problem Definition

A. Background

The workflow of modern browsers is usually very complicated, due to the abundant web features to support. Figure 1 shows a simplified architecture of the Firefox browser. Other browsers' basic architectures are similar in general.

After parsing the HTML, CSS and multimedia input from Internet, the Firefox Gecko engine builds a `DOM` (Document Object Model) representation to model the HTML documents. Its `layout engine` computes the page's layout based on the DOM and CSS information, and then the `rendering engine` renders the web page and present it to users.

The SpiderMonkey JavaScript engine can execute the scripts in the web page and interact with page elements through the DOM interface. The `interpreter` interprets the script statements one by one. For JavaScript functions that will be executed multiple times, interpretation is slow. As a result, SpiderMonkey invokes the IonMonkey `JIT compiler` to compile them to native executable code, and diverts further invocation of these JavaScript functions to the JIT compiled code. The `instruction generator` is responsible for platform-specific native instruction generation, and the `JIT code installer` writes the compiled JavaScript functions to the JIT code memory, and patch the instructions that have absolute addresses operands.

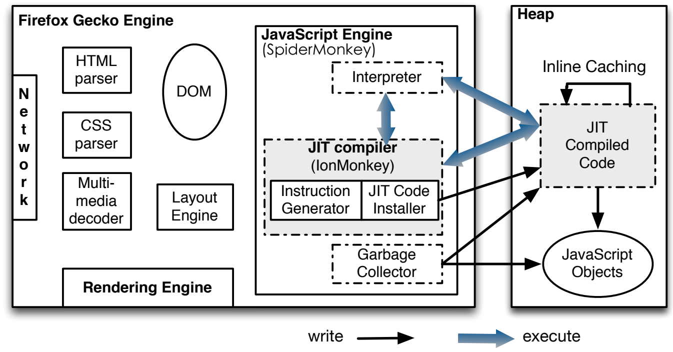

Fig. 1: Architecture of the Firefox Gecko Engine.

In addition, there is a `garbage collector` in the JavaScript engine, responsible for the garbage collection [42] of JavaScript objects and code. It may also modify the JIT code at runtime. Moreover, the JIT compiler may use optimization techniques such as `inline caching` [43, 44], which modify the JIT code as well.

B. Threat Model

We assume attackers have the following capabilities: (1) attackers can write to any writable memory, and they can corrupt control data such as return addresses and function pointers; (2) attackers can read arbitrary mapped memory, and thus they can launch information leakage attacks and bypass secret-based defenses such as ASLR; (3) attackers cannot directly read or write to registers, and they can only achieve this indirectly by using existing instructions that propagates data between registers and memory.

This assumption is realistic. In real world attacks, attackers are able to exploit vulnerabilities to obtain these capabilities, especially for applications supporting scripting languages and JIT compilation.

On the defense side, we assume (1) popular defenses such as ASLR and W⊕X are deployed; (2) the theoretical threat non-control-data attacks [45] that may lead to control-flow hijacking are out of this paper's scope.

C. Security Risks

In addition to traditional control-flow hijacking threats, for web browsers, there are several new and more critical threats.

1) Risks Brought by Scripting Languages: Modern browsers all support scripting languages (e.g., JavaScript). Scripting languages make the web more dynamic, and make user experiences better. At the same time, however, it also brings some risks to users.

- *Heap Spraying.* With scripting languages, attackers are able to allocate a lot of memory indirectly. By feeding special scripts to browsers, attackers can spray a lot of objects in the heap, called *heap spraying* [11], and make some of them take the expected memory address. In this way, attackers can predicate some objects' addresses and bypass the ASLR defense.
- *Information Leakage.* By writing scripts, it is much easier for attackers to exploit vulnerabilities to read memory, and retrieve the leaked value for further use. The study [28] discussed many different types of information leakage attacks.
- *Shellcode Generation.* Script languages provide another way for attackers to build expected shellcode, even without the help of advanced attacks such as ROP that is used in statically compiled code, to bypass W⊕X. The study [13] presents a way to write shellcode with scripts.

2) Risks Brought by JIT Compilation: Major web browsers all deploy the JIT compilation optimization technique to improve the JavaScript performance. This optimization also introduces security risks to web users.

- *JIT Code Corruption and Injection.* As the JIT code memory is both executable and writable, the classic W⊕X defense is not applicable. Attackers can thus corrupt the JIT compiled code, or inject code to the JIT memory [46].
- *JIT Code Reuse.* Attackers can also manipulate the layout of the JIT memory by launching attacks such as JIT spraying [14–17], and then reuse the JIT compiled code to build the shellcode.

D. Challenges

When defending web browsers against control-flow hijacking attacks, there are several challenges in practice.

1) Lack of W⊕X: Due to the requirements of inline caching, garbage collection, and the like, the JIT compiled code is both executable and writable, rendering W⊕X inapplicable.

Without W⊕X, attackers can bypass any deployed security solutions. For example, if a CFI solution is deployed without the support of W⊕X, then attackers can directly overwrite the instrumented CFI security checks, totally disabling the CFI enforcement.

2) Lack of CFG: Because the JIT compiled code is generated at runtime, there is no static control-flow graph (CFG) information available. As a result, solutions such as CFI cannot be deployed to the JIT compiled code directly.

IV. Design

In this section, we discuss the design of `JITScope`, briefly explaining the policies and how they are enforced on browsers.

A. Security Primitives

An effective defense should enforce the following primitives.

- *Statically compiled code cannot transfer control flow to illegal targets.* Control flow should only transfer to targets intended at compile-time; only occasionally should it be transferred to JIT compiled code.
- *JIT compiled code cannot transfer control flow to illegal targets.* Control flow should only transfer to legitimate JIT code entries, except in a few cases in which it can be transfered to fixed targets in statically compiled code.
- *JIT compiled code cannot be tampered with.* Otherwise, attackers can disable this defense by overwriting the code.

In this way, all control flows in statically compiled code and JIT compiled code are restricted, especially the flow between statically compiled code and JIT compiled code. Furthermore, these security enforcements are protected from tampering.

In addition, a practical solution should provide better security enforcement while incurring a low performance overhead.

B. Overview of JITScope

`JITScope` enforces two security policies for web browsers. First, it enforces the W⊕X policy on JIT compiled code. As W⊕X has already been adopted by the operating system, the statically compiled code is already under protection. Once the W⊕X policy is extended to the JIT compiled code, attackers cannot corrupt or inject any code into applications.

Second, `JITScope` enforces the CFI policy on both statically compiled code and JIT compiled code. For statically compiled code, we deploy a fine-grained CFI policy based on functions' type information. For JIT compiled code, there is no control flow graph information or type information available. Thus, a coarse-grained CFI policy is deployed for JIT compiled code.

Traditional CFI solutions will check the targets of all indirect control transfer instructions, including indirect call or jump instructions, and the return instructions. We find that the CFI solutions on return instructions are still slow, and use another effective and efficient solution to protect return instructions, i.e., shadow stack [47].

C. W⊕X for JIT Compiled Code

As shown in Figure 2, `JITScope` introduces three delegates in the JavaScript engine to enforce W⊕X on JIT compiled code, i.e., the `fwd-exec`, `bwd-exec` and `write` delegates. The first two delegates are responsible for setting the memory permission before executing JIT code, and the last one takes care of the write operations to JIT code memory.

All write operations to the JIT code, including the operations triggered by JIT code installation, inline caching and garbage collection, are enforced by `JITScope` to dispatch through the `write delegate`. This delegate enables the writable permission of the JIT code memory, but drops the executable permission at the same time. Once the write operation finishes, it turns off the writable permission of the JIT code memory immediately. In general, there is only a small

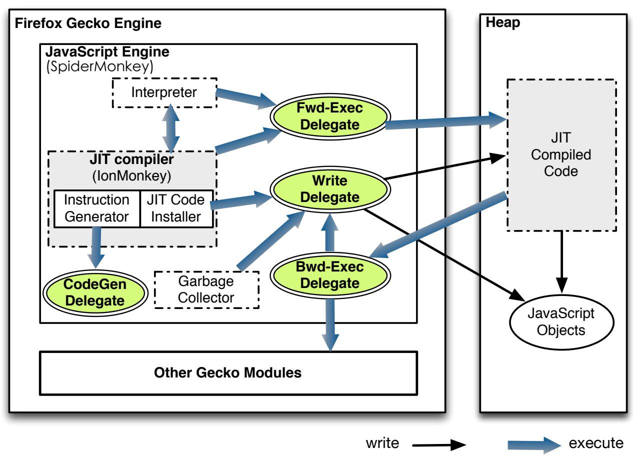

Fig. 2: Illusion of `JITScope` for the JIT compiled code.

time window that the JIT code memory is writable. Moreover, `JITScope` only enables the writable permission for a single memory page that needs to be written. As a result, it is very difficult for attackers to overwrite the JIT code memory, even by utilizing race condition attacks.

All function calls to JIT compiled code are enforced to dispatch through the `fwd-exec delegate`. This delegate sets the target memory to read-only and executable, before calling the JIT code. The JIT compiled code may also call some functions (called *VM functions* in Firefox) provided by the JavaScript engine, or even by the browser. These function calls are all dispatched through our `bwd-exec delegate`. This delegate only allows legitimate VM function calls, and also sets the target memory to read-only and executable, before the VM functions return to the JIT compiled code.

D. CFI for Statically Compiled Code

`JITScope` enforces the CFI policy on statically compiled code. It utilizes the type information from the browser's source code, and deploys fine-grained CFI on the statically compiled code. In particular, before each function, `JITScope` instruments an ID computed from this function's type information. And before each indirect call and jump instruction, `JITScope` instruments a CFI security check to match the transfer target's ID against the expected ID computed from the expected function's type information.

E. CFI for JIT Compiled Code

The JIT compiled code is generated at runtime (and there is no CFG information available) so we only enforce a coarse-grained CFI policy on the JIT compiled code. As shown in Figure 2, we introduce a `CodeGen delegate` in the JavaScript engine to deploy the CFI policy. This delegate first randomly selects an ID at compile-time. When the JIT compiler is going to generate a function, the `CodeGen delegate` instruments the ID before this function. When the JIT compiler is going to generate an indirect call or jump instruction, its `instruction generator` invokes the `CodeGen delegate` to instrument CFI security checks before the call or jump. The checks validate the existence of the ID before the target function to ensure the transfer target is a valid JIT function entry.

F. Shadow Stack for Return Instructions

`JITScope` only applies the CFI policy on indirect call and jump instructions. For the return instructions, we use the shadow stack solution to provide a better performance. At each function entry, `JITScope` copies the return address of current function frame to a shadow stack. Then at the function exit, it matches the return address on the original stack against the one on the shadow stack. `JITScope` reports a security violation if a mismatch occurs. This solution provides a more accurate target validation than CFI solutions, with a lower performance overhead.

For JIT compiled code, we also utilize the `CodeGen delegate` to instrument the shadow stack related operations to runtime generated return instructions.

In addition to the basic shadow stack solution, we improve its security and reliability in several ways. First, we put the shadow stack in a separate memory region, indexed by a dedicated segment register (on the x86 platform). Because no normal instructions will access memory through the dedicated segment register, attackers cannot tamper with the shadow stack. Second, we build a separate shadow stack for each thread by using thread-local storage. In this way, the shadow stack is thread-safe.

G. Compatibility Issues

`JITScope` will instrument security checks to the web browser, to validate indirect control transfer instructions' targets at runtime. However, the web browser may indirectly call external functions that are not defined in the browser, such as functions provided by the operating system or libraries. There are no IDs instrumented before these functions, and, as a result, the CFI checks instrumented by `JITScope` will fail, causing false positives.

In order to make the web browser hardened by `JITScope` compatible with the operating system and libraries, we also introduce a wrapper library for these target functions. In this wrapper library, for each target function, there is a wrapper function that will eventually invoke the original target function. `JITScope` will instrument an ID before this wrapper function based on its type information. In this way, the CFI checks in the browser can work seamlessly, even if the transfer targets are external functions.

V. Implementation

Figure 3 shows the basic workflow of `JITScope`. Briefly, it instruments the JavaScript engine with the `CodeGen delegate` that is responsible for instrumenting CFI enforcements to the JIT compiled code, as well as three other delegates for enforcing W$\oplus$X to the JIT memory. It also provides a wrapper library to wrap all external functions indirectly called by the web browser. Finally, the source code

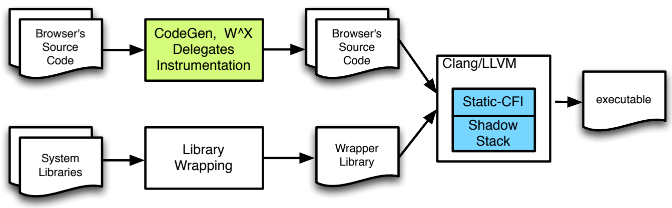

Fig. 3: Implementation of `JITScope`.

of the browser is compiled by the Clang compiler [22], and then linked with the wrapper library. The CFI enforcement for statically compiled code, and the shadow stack solution are all implemented as analysis passes in the LLVM framework. The output of the compiler is the final executable browser.

We implement the prototype of `JITScope` on the popular Firefox web browser. In this section, we discuss the details of the implementation.

A. Modification to the JavaScript Engine

`JITScope` modifies the source code of the SpiderMonkey JavaScript engine to enforce the W⊕X policy, CFI policy, and shadow stack policy on JIT compiled code. As discussed earlier, `JITScope` introduces four delegates to the JavaScript engine, i.e., the `fwd-exec`, `bwd-exec`, `write` and `CodeGen` delegates.

1) Enforce W⊕X policy: In SpiderMonkey, all transitions from the statically compiled code to the JIT compiled code are made through the API `jit::IonCannon` and `jit::EnterBaselineMethod`. So, our `fwd-exec delegate` is built upon these two APIs. It drops the writable permission of the target JIT code memory before entering target JIT code, by using the API provided by the operating system, e.g., `mprotect`.

The transition from the JIT compiled code to the statically compiled code is made through *VM functions*. For each statically compiled VM function, the JIT compiler generates a wrapper function in the JIT code memory. At runtime, the JIT code can only call these VM wrapper functions, and these VM wrapper functions directly call the associated statically compiled VM functions. As a result, we deploy the `bwd-exec delegate` in the VM functions to drop the writable permission before returning to the JIT compiled code.

For the `write delegate`, we identify all write operations to the JIT compiled code in SpiderMonkey. And wrap all these write operations with the `write delegate`. This delegate drops the executable permission and enables the writable permission. After the write operation finishes, it immediately turns the JIT memory back to executable only.

2) Enforce CFI policy: In SpiderMonkey's JIT compiler IonMonkey, there is an instruction generator responsible for generating platform-specific native code. Our `CodeGen delegate` is built upon this instruction generator, to enforce the coarse-grained CFI and the shadow stack solution on JIT compiled code. Once the JIT compiler is going to generate a JIT function, i.e., when the procedure `JSC::ExecutableAllocator::alloc()` is invoked, this delegate instruments the predefined ID before this function. Once the compiler is going to generate an indirect call or jump instruction, i.e., when the procedure `JSC::X86Assembler::call()` is invoked, the delegate instruments a check to validate the existence of the ID before the target function.

3) Enforce shadow stack policy: In addition to the CFI policy, the `CodeGen delegate` also instruments the JIT compiled code to support the shadow stack. In particular, it adds a special function call at the beginning of the JIT compiled function, to push the current return address to the thread's shadow stack. It also adds another function call at the end of this JIT compiled function, to match the return address between the original stack and the shadow stack and pop the shadow stack.

B. Analysis Pass based on LLVM

We use several LLVM analysis passes to enforce the security policies on statically compiled code.

First, we use an analysis pass to enforce a fine-grained CFI on statically compiled code. This pass analyzes each function in the Intermediate Representation (IR) level, and then iterate over all the instructions. For each indirect call or jump instruction, it instruments the CFI security check before this instruction. More specifically, it computes the expected ID based on the target function's type information, and then adds a check instruction to match this expected ID against the ID from target function. This security check validates the transfer target at runtime. We also modify LLVM's `CodeGen` backend to instrument an ID before each generated function based on its type information.

Another analysis pass then deploys the shadow stack solution to protect the return instructions in statically compiled code. This pass also analyzes each IR-level function. It adds a function call at the beginning and the end of each function, to push and pop return addresses to the shadow stack, and match the return addresses to detect security violations.

C. Library Wrapping

For external functions indirectly called in browsers, wrapper functions are introduced to eliminate compatibility issues.

For each candidate function (e.g., *foo*), we generate a wrapper function (e.g., *__wrap_foo*). This wrapper function jumps to the original function directly. All these wrapper functions are put into one source file, and are then compiled by Clang with our analysis pass. As a result, the expected ID is instrumented before each wrapper function. Finally, when compiling the browser with Clang, we utilize the "–wrap" option of the GNU linker `ld`, to automatically replace references to any target external function (e.g. *foo*) with its wrapper function (e.g., *__wrap_foo*). This eliminates all compatibility issues.

It is important to note that there is a special library function `dlsym`. This function resolves the address of a target function

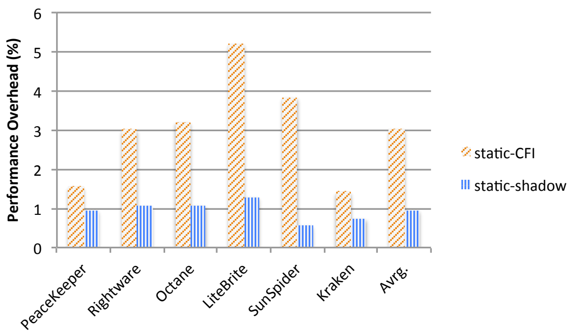

Fig. 4: Performance overhead of Firefox when `JITScope` only protects statically compiled code.

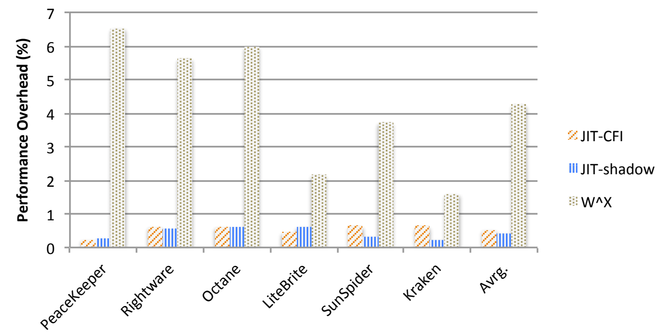

Fig. 5: Performance overhead of Firefox when `JITScope` only protects JIT compiled code.

at runtime, and return this address to the caller function. If the browser indirectly calls this function to resolve an external function, and then indirectly invokes the target external function, it still causes a compatibility issue. To deal with this special case, we provide a special wrapper function for the `dlsym`, i.e., `__wrap_dlsym`. This wrapper function `__wrap_dlsym` resolves the target function's address at runtime, creates a temporary code snippet for the target function, and instruments the code snippet with its expected ID. The wrapper function `__wrap_dlsym` then returns a pointer to this temporary code snippent as the return value to the browser.

VI. Evaluation

`JITScope` is built on LLVM 3.4, has about 800 lines of C++ code for the analysis passes and another 300 lines of Python scripts to wrap external libraries, and only modifies about 200 lines of source code of Firefox. We evaluate its performance and security on a system with x86-64 Ubuntu 12.04, an Intel Core i7-2600 CPU at 3.4GHz, and 8GB of physical memory.

A. Performance Evaluation

We test the `JITScope`-hardened Firefox's performance on six popular browser benchmarks, including Google's Octane [48], Mozilla's Kraken [49], Apple's Sunspider [50], Microsoft's LiteBrite [51], RightWare [52] and PeaceKeeper [53]. These benchmarks measure different aspects of a browser, from the speed of JavaScript handling, HTML rendering and HTML5 support, to the JIT compiler's latency.

1) Evaluate `JITScope` on Statically Compiled Code: Figure 4 shows the performance overhead of Firefox, when `JITScope` only protects the statically compiled code. More specifically, `JITScope` applies the CFI policy and the shadow stack policy on the statically compiled code separately. In other words, the two LLVM analysis passes for CFI and shadow stack are deployed, and others are not.

As this figure shows, when the CFI policy is separately deployed on statically compiled code (i.e., static-CFI in the figure), the average performance overhead is about 3.04%. The minimum performance overhead is about 1.42% (i.e., the Kraken benchmark), whereas the maximum overhead is about 5.21% (i.e., the LiteBrite benchmark).

If the shadow stack policy is separately deployed on statically compiled code (i.e., static-shadow in the figure), the average performance overhead is about 0.95%. The minimum is about 0.58% (i.e., the SunSpider benchmark), whereas the maximum is about 1.29% (i.e., the LiteBrite benchmark).

2) Evaluate `JITScope` on JIT Compiled Code: Figure 5 shows the performance overhead of Firefox, when `JITScope` only protects the JIT compiled code. More specifically, `JITScope` applies the CFI policy (with the `CodeGen delegate`) and the shadow stack policy (with the `CodeGen delegate`) on the JIT compiled code separately, and extends the W⊕X policy to the JIT compiled code (with the delegates `fwd-exec`, `bwd-exec` and `write`).

As Figure 5 shows, when the CFI policy is separately deployed on JIT compiled code (i.e., JIT-CFI in the figure), the average performance overhead is about 0.54%, and the minimum and maximum overheads are 0.24% and 0.64% respectively.

If the shadow stack policy is separately deployed on JIT compiled code (i.e., JIT-shadow in the figure), the average performance overhead is about 0.43%, and the minimum and maximum overheads are 0.24% and 0.62% respectively.

If the W⊕X policy is separately deployed on JIT compiled code (i.e., W^X in the figure), the average performance overhead is about 4.26%, and the minimum and maximum overheads are 1.60% and 6.53% respectively.

3) Evaluate `JITScope` on All Code: Figure 6 shows the overall performance overhead of the `JITScope`-hardened Firefox. In particular, if all security checks (i.e., CFI and shadow stack) are deployed on statically compiled code, the average performance overhead is about 4.02%. If all security checks (i.e., CFI, shadow stack and W⊕X) are deployed on JIT compiled code, the average overhead is about 5.28%.

If `JITScope` deploys all security policies (on both the statically- and JIT-compiled code of Firefox), the average performance overhead is about 9.51%, the minimum overall overhead is about 4.75% (i.e., Kraken), whereas the maximum overall overhead is about 11.93% (i.e., Octane).

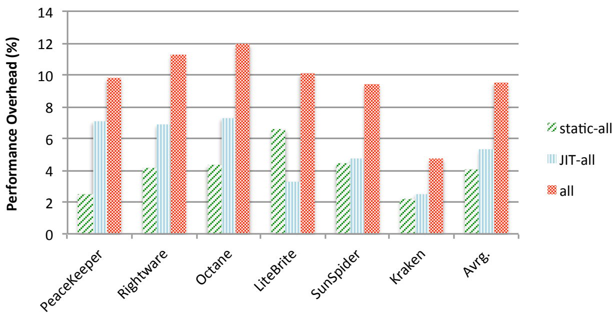

Fig. 6: Performance overhead of Firefox when `JITScope` protects all code.

4) Performance Analysis: The performance overhead of the `JITScope`-hardened Firefox is brought by the security checks instrumented before indirect control transfer instructions (i.e., instrumented by the CFI and shadow stack policy for both statically and JIT compiled code), and by the runtime memory page permission switching (i.e., instrumented by the W⊕X policy for the JIT compiled code).

As shown in Figure 4, 5 and 6, the W⊕X policy introduces most of the performance overhead. For example, for the PeaceKeeper benchmark, the W⊕X policy introduces about 6.5% overhead, while the overall performance overhead is about 9.78%. This high performance overhead is due to the frequent memory page permission changes. `JITScope` uses the system API `mprotect` to turn on or turn off the writable and executable permission of the target memory. This API traps to the kernel at runtime, and invalidates the translation lookaside buffer (TLB). As a result, there are many context switches between user space and the kernel space, as well as many heavy TLB installations, introducing a lot of overhead.

Another interesting result is that, the hardened Firefox has the smallest performance overhead on the Kraken benchmark developed by Mozilla, perhaps because Mozilla has implemented some optimizations in Firefox for this specific benchmark.

B. Protection Effectiveness Analysis

`JITScope` enforces a fine-grained CFI on statically compiled code, and extends the CFI to JIT compiled code, prevents indirect call and jump instructions from jumping to targets of illegal types. Moreover, it deploys the shadow stack on both statically and JIT compiled code, providing an accurate return targets match at runtime. It blocks return instructions from jumping to any illegal target, even if the target has a correct type and is allowed by the traditional CFI.

The measurement AIR (Average Indirect target Reduction ratio [8]) reflects how many invalid control transfers can be blocked by a defense solution. For `JITScope`, the AIR ratio is about 99.98%.

We also evaluated the `JITScope`-hardened Firefox against real world exploits. As there is no public available exploits for the latest Firefox in Linux, we thus simulate an attack scenario. First, we introduce two `JS_Native` APIs into the Firefox's source code. These two APIs can be directly invoked by user's JavaScript functions, to read and write arbitrary memory address, simulating the arbitrary memory read and write vulnerabilities that can be exploited by attackers. Then, we launch the exploit described in [54] to attack `JITScope`-hardened Firefox. This exploit launches a heap spray first to manipulate the memory layout, and then overwrites the virtual table pointer of objects in the sprayed memory. Finally, when the objects' virtual function is invoked, the control flow is hijacked. The experiment shows that, the `JITScope`-hardened Firefox successfully blocks this attack at runtime.

`JITScope` uses `mprotect` to switch the property of JIT code memory, is thus subject to race condition attacks. For example, attackers may overwrite the JIT code memory in another thread while this memory is set to writable. However, this risk is very low because the attack time window is very small. `JITScope` turns off the writable property immediately after the legitimate write operation finishes.

VII. Conclusion

Just-In-Time compilation is now widely adopted by modern applications, especially web browsers. Traditional control-flow integrity solutions provide no protection against the JIT compiled code. We propose a general solution `JITScope` to prevent the JIT compiled code from being exploited. `JITScope` enforces a general CFI policy on both statically compiled code and JIT compiled code, and also enforces the W⊕X policy on JIT compiled code. It therefore provides a strong protection for JIT code, and can defeat most control-flow hijacking attacks. Experiments show that this solution has a reasonable performance overhead, and can be deployed in practice to defend against real-world exploits.

Acknowledgment

This research was supported in part by the Natural Science Foundation award CCF-0424422, DARPA award HR0011-12-2-005, and FORCES (Foundations Of Resilient CybEr-Physical Systems), which receives support from the National Science Foundation (NSF award numbers CNS-1238959, CNS-1238962, CNS-1239054, CNS-1239166).

References

[1] N. Asokan, V. Niemi, and K. Nyberg, "Man-in-the-middle in tunnelled authentication protocols," in *Security Protocols*. Springer, 2005, pp. 28–41.
[2] K. Spett, "Cross-site scripting," *SPI Labs*, pp. 1–20, 2005.
[3] R. Auger, "The cross-site request forgery (csrf/xsrf) faq," *CGISecurity. com. Apr*, vol. 17, 2008.
[4] PaX Team, "PaX address space layout randomization (ASLR)," http://pax.grsecurity.net/docs/aslr.txt, 2003.
[5] S. Andersen and V. Abella, "Data Execution Prevention: Changes to Functionality in Microsoft Windows XP Service Pack 2, Part 3: Memory Protection Technologies," http://technet.microsoft.com/en-us/library/bb457155.aspx, 2004.
[6] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity," in *ACM Conference on Computer and Communications Security (CCS)*, 2005.
[7] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou, "Practical control flow integrity and randomization

for binary executables," in *Security and Privacy (SP), 2013 IEEE Symposium on*. IEEE, 2013, pp. 559–573.
[8] M. Zhang and R. Sekar, "Control flow integrity for cots binaries." in *USENIX Security*, 2013, pp. 337–352.
[9] Z. Wang and X. Jiang, "HyperSafe: A lightweight approach to provide lifetime hypervisor control-flow integrity," in *IEEE Symposium on Security and Privacy*, 2010.
[10] B. Niu and G. Tan, "Modular control-flow integrity," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 2014, p. 58.
[11] M. Daniel, J. Honoroff, and C. Miller, "Engineering heap overflow exploits with JavaScript," in *Workshop on Offensive Technologies (WOOT)*, 2008.
[12] P. Vreugdenhil, "Pwn2own 2010 windows 7 internet explorer 8 exploit," http://vreugdenhilresearch.nl/Pwn2Own-2010-Windows7-InternetExplorer8.pdf, 2010.
[13] Y. Yu, "Rops are for the 99%," in *The 14th Annual CanSecWest Conference*, 2014.
[14] D. Blazakis, "Interpreter exploitation." in *WOOT*, 2010.
[15] D. Blazakis, "Interpreter exploitation: Pointer inference and jit spraying," *BlackHat DC*, 2010.
[16] W. Lian, H. Shacham, and S. Savage, "Too LeJIT to Quit: Extending JIT Spraying to ARM," in *the Network and Distributed System Security Symposium (NDSS)*, 2015.
[17] M. Athanasakis, E. Athanasopoulos, M. Polychronakis, G. Portokalidis, and S. Ioannidis, "The Devil is in the Constants: Bypassing Defenses in Browser JIT Engines," in *the Network and Distributed System Security Symposium (NDSS)*, 2015.
[18] A. Homescu, S. Brunthaler, P. Larsen, and M. Franz, "librando: Transparent code randomization for just-in-time compilers," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM, 2013, pp. 993–1004.
[19] T. Wei, T. Wang, L. Duan, and J. Luo, "Insert: Protect dynamic code generation against spraying," in *Information Science and Technology (ICIST), 2011 International Conference on*. IEEE, 2011, pp. 323–328.
[20] J. Ansel, P. Marchenko, Ú. Erlingsson, E. Taylor, B. Chen, D. L. Schuff, D. Sehr, C. L. Biffle, and B. Yee, "Language-independent sandboxing of just-in-time compilation and self-modifying code," *ACM SIGPLAN Notices*, vol. 46, no. 6, pp. 355–366, 2011.
[21] B. Niu and G. Tan, "Rockjit: Securing just-in-time compilation using modular control-flow integrity," in *Proceedings of the 21st ACM Conference on Computer and Communications Security*. ACM, 2014.
[22] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*. IEEE, 2004, pp. 75–86.
[23] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton, "StackGuard: Automatic adaptive detection and prevention of buffer-overflow attack," in *USENIX Security Symposium*, 1998.
[24] D. Jang, Z. Tatlock, and S. Lerner, "SAFEDISPATCH: Securing C++ Virtual Calls from Memory Corruption Attacks," in *20th Annual Network and Distributed System Security Symposium*, 2014.
[25] C. Zhang, C. Song, K. Z. Chen, Z. Chen, and D. Song, "VTint: Protecting Virtual Function Tables' Integrity," in *the Network and Distributed System Security Symposium (NDSS)*, 2015.
[26] H. Shacham, M. Page, B. Pfaff, E.-j. Goh, N. Modadugu, and D. Boneh, "On the effectiveness of address-space randomization," *Proceedings of the 11th ACM conference on Computer and communications security (CCS'04)*, p. 298, 2004.
[27] H. Shacham, "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)," in *ACM Conference on Computer and Communications Security (CCS)*, 2007.
[28] F. J. Serna, "The info leak era on software exploitation," in *Blackhat USA*, 2012.
[29] E. Göktas, E. Athanasopoulos, H. Bos, and G. Portokalidis, "Out of control: Overcoming control-flow integrity," in *IEEE S&P*, 2014.
[30] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, Ú. Erlingsson, L. Lozano, and G. Pike, "Enforcing forward-edge control-flow integrity in gcc & llvm," in *23rd USENIX Security Symposium (USENIX Security 14)*. San Diego, CA: USENIX Association, Aug. 2014, pp. 941–955. [Online]. Available: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/tice
[31] R. Wahbe, S. Lucco, T. Anderson, and S. Graham, "Efficient software-based fault isolation," in *Symposium on Operating Systems Principles (SOSP)*, 1994.
[32] J. Seward and N. Nethercote, "Using valgrind to detect undefined value errors with bit-precision." in *USENIX Annual Technical Conference, General Track*, 2005, pp. 17–30.
[33] D. Dhurjati and V. Adve, "Backwards-compatible array bounds checking for c with very low overhead," in *Proceedings of the 28th international conference on Software engineering*. ACM, 2006, pp. 162–171.
[34] H. Patil and C. Fischer, "Low-cost, concurrent checking of pointer and array accesses in c programs," *Softw., Pract. Exper.*, vol. 27, no. 1, pp. 87–110, 1997.
[35] W. Xu, D. C. DuVarney, and R. Sekar, "An efficient and backwards-compatible transformation to ensure memory safety of c programs," in *ACM SIGSOFT Software Engineering Notes*, vol. 29, no. 6. ACM, 2004, pp. 117–126.
[36] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic, "SoftBound: highly compatible and complete spatial memory safety for C," in *Conference of Programming Language Design and Implementation (PLDI)*, 2009.
[37] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic, "CETS: compiler enforced temporal safety for C," in *International Symposium on Memory Management (ISMM)*, 2010.
[38] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song, "Code-pointer integrity," *OSDI'14*, 2014, 00000. [Online]. Available: https://www.usenix.org/system/files/conference/osdi14/osdi14-paper-kuznetsov.pdf?utm_source=dlvr.it&utm_medium=tumblr
[39] C. Song, C. Zhang, T. Wang, W. Lee, and D. Melski, "Exploiting and protecting dynamic code generation," in *the Network and Distributed System Security Symposium (NDSS)*, 2015.
[40] P. Chen, R. Wu, and B. Mao, "Jitsafe: a framework against just-in-time spraying attacks," *IET Information Security*, vol. 7, no. 4, pp. 283–292, 2013.
[41] P. Chen, Y. Fang, B. Mao, and L. Xie, "Jitdefender: A defense against jit spraying attacks," in *Future Challenges in Security and Privacy for Academia and Industry*. Springer, 2011, pp. 142–153.
[42] H.-J. Boehm and M. Weiser, "Garbage collection in an uncooperative environment," *Software: Practice and Experience*, vol. 18, no. 9, pp. 807–820, 1988.
[43] L. P. Deutsch and A. M. Schiffman, "Efficient implementation of the smalltalk-80 system," in *Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. ACM, 1984, pp. 297–302.
[44] U. Hölzle, C. Chambers, and D. Ungar, "Optimizing dynamically-typed object-oriented languages with polymorphic inline caches," in *ECOOP'91 European Conference on Object-Oriented Programming*. Springer, 1991, pp. 21–38.
[45] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer, "Non-control-data attacks are realistic threats," in *USENIX Security Symposium*, 2005.
[46] Pinkie Pie, "Mobile Pwn2Own Autumn 2013 - Chrome on Android - Exploit Writeup," https://docs.google.com/document/d/1tHElG04AJR5OR2Ex-m_Jsmc8S5fAbRB3s4RmTG_PFnw/edit, 2013.
[47] Vendicator, "A "stack smashing" technique protection tool for Linux," http://www.angelfire.com/sk/stackshield/, 2000.
[48] Google, "Octane JavaScript benchmark suite," https://developers.google.com/octane/, 2014.
[49] Mozilla, "Kraken 1.1 javascript benchmark suite," http://krakenbenchmark.mozilla.org/, 2014.
[50] Apple, "Sunspider 1.0.2 javascript benchmark suite," https://www.webkit.org/perf/sunspider/sunspider.html, 2014.
[51] Microsoft IE, "LiteBrite: HTML, CSS and JavaScript Performance Benchmark," http://ie.microsoft.com/testdrive/Performance/LiteBrite/, 2014.
[52] RightWare, "Browsermark 2.1 benchmark," http://browsermark.rightware.com/, 2014.
[53] FutureMark, "Peacekeeper: HTML5 browser speed test," http://peacekeeper.futuremark.com/, 2014.
[54] P. Argyroudis and C. Karamitas, "Exploiting the jemalloc memory allocator: Owning firefoxÂ¡Â¯s heap patroklos," in *BlackHat USA*, 2012.