Exploiting and Protecting Dynamic Code Generation

Chengyu Song
Georgia Institute of Technology
csong84@gatech.edu

Chao Zhang
UC Berkeley
chaoz@berkeley.edu

Tielei Wang, Wenke Lee
Georgia Institute of Technology
tielei.wang@gmail.com, wenke@cc.gatech.edu

David Melski
GrammaTech
melski@grammatech.com

***Abstract*—Many mechanisms have been proposed and deployed to prevent exploits against software vulnerabilities. Among them, W⊕X is one of the most effective and efficient one. W⊕X prevents memory pages from being writable and executable at the same time. This enforcement rendered the decades old shellcode injection technique infeasible.**

In this paper, we demonstrate that the traditional shellcode injection attack can be revived through a code cache injection technique. Specifically, dynamic code generation, a technique widely used in just-in-time (JIT) compilation and dynamic binary translation (DBT), generates and modifies code on the fly, in order to promote performance or security. The dynamically generated code fragments are stored in code cache, which is writable and executable either at the same time or alternately, resulting in an opportunity for exploit. This threat is especially realistic when the generated code is multi-threaded, because simply switching between writable and executable leaves a time window for exploit. To illustrate this threat, we have crafted a proof-of-concept exploit against modern browsers that support *Web Workers*.

To mitigate this code cache injection threat, we propose a new dynamic code generation architecture. This new architecture relocates the dynamic code generator to a separate process, in which the code cache is writable. But in the original process where the generated code executes, the code cache remains read-only. The code cache is synchronized across the writing process and the execution process through shared memory. And interaction between the code generator and the generated code is handled transparently through remote-procedure call (RPC). We have ported Google V8 JavaScript engine and Strata DBT to this new architecture. Our implementation experience showed that the engineering effort for porting to this new architecture is very small. Evaluation of our prototype implementation showed that this new architecture can defeat the code cache injection attack with small performance overhead.

I. Introduction

Exploits against software vulnerabilities remain one of the most severe threats to cyber security. To mitigate this threat, many techniques have been proposed, including data execution prevention (DEP) [1] and address space layout randomization (ASLR) [2], which have been widely deployed and very successful. DEP is a subset of the more general security policy W⊕X, which enforces that memory should either be writable but not executable (e.g., data segments), or be executable but read-only (e.g., code segments). This enforcement can completely mitigate traditional exploits that inject malicious shellcode into data segments. Consequently, attackers have to leverage more complicated exploit techniques, such as return-to-libc [3] and return-oriented-programming (ROP) [4]. Moreover, W⊕X memory has become the foundation of many other protection techniques, such as control flow integrity (CFI) [5], [6], [7].

However, the effectiveness of W⊕X can be undermined by another important compilation technique – dynamic code generation (DCG). With the ability to generate and execute native machine code at runtime, DCG is widely used in JIT compilers [8] and dynamic binary translators (DBT) [9], [10], [11] to improve performance, portability, and security. For example, JIT compilers for dynamic languages (e.g., JavaScript and ActionScript) can leverage platform information and runtime execution profile information to generate faster native code. And DBTs can leverage DCG to provide dynamic analysis capability [11], cross-platform or cross-architecture portability [12], [13], bug diagnostic [14], [15], and better security [16], [17], [18], [19], [20].

A fundamental challenge posed by DCG is that, the code cache, in which the dynamically generated code is stored, needs to be both writable (for code emitting, code patching and garbage collection) and executable. This violates the W⊕X policy and enables a new attack vector. We have observed a real world exploit that delivers shellcode into the writable code cache and successfully compromises the Chrome web browser [21].

Solving this problem seems trivial. A straightforward idea, which has been adopted in browsers like mobile Safari, is demonstrated in Figure 1. This technique keeps the code cache as read-only and executable (`RX`) when the generated code is executing; switches to writable but not executable (`WR`) when it needs to be modified ($t1$); and switches back to `RX` when the write operation finishes ($t2$). As a result, the code cache will remain read-only when the generated code is executing; and the attack demonstrated in [21] can thus be mitigated.

Unfortunately, in addition to performance overhead, this simple mechanism does not work well with multi-threaded programs. First, if the code generator uses a shared code cache for all threads (e.g., PIN [11]), then obviously the code cache cannot be switched to `WR`, because other concurrently running threads require the executable permission. Second, even if the code generator uses dedicated code cache for each thread (e.g., JS engines), the protection is still flawed and is subject to *race condition* attacks [22], as shown in Figure 2. More specifically, memory access permissions are applied to the whole process and are shared among all threads. When one thread, say thread A, turns on the writable permission for its code cache (e.g., for code emitting), the code cache also becomes writable to all other threads. Once the protection is gone, another concurrently running thread, thread B, can (maliciously) overwrite thread A's code cache to launch attacks. This is similar to the classic time-to-check-time-to-use (TOCTOU) attacks [23], where the resource to be accessed is modified between the check and the use, by exploiting race conditions.

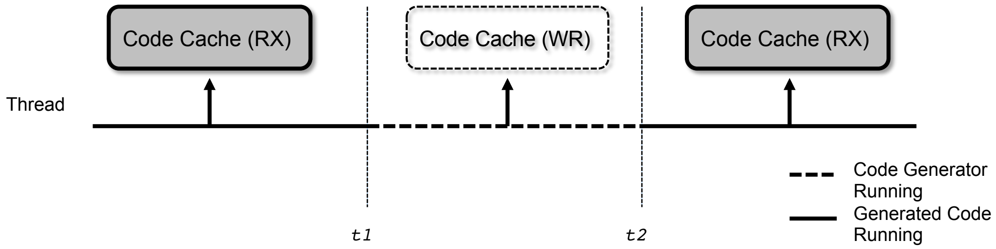

Fig. 1: A permission switching based W⊕X enforcement. The code cache is kept as read-only when the generated code is executing. When the code generator is invoked ($t1$), the permission is changed to writable; and when the generator finishes its task ($t2$), the permission is changed back to read-only.

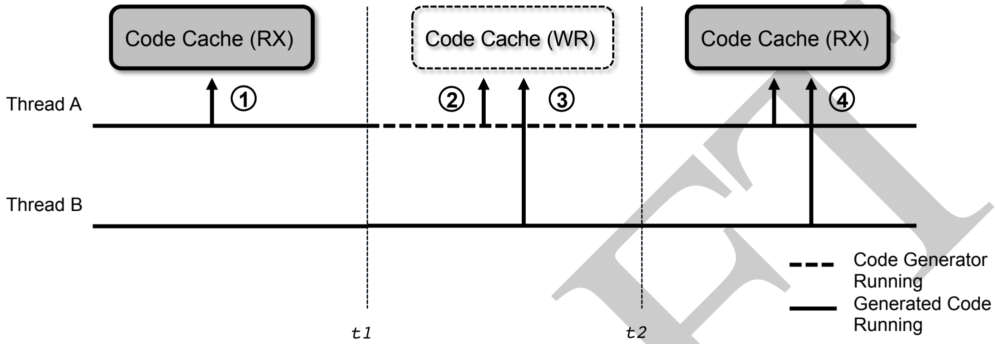

Fig. 2: Race-condition-based attack using two threads. With switching based W⊕X enforcement, a single thread (A) can no longer attack the code cache (access 1). But the code cache can still be attacked using multiple threads. As when the code generator is serving one thread (access 2), the code cache will also become writable for other thread (access 3). The attack window refers to $t2$ - $t1$, as once the code generator finishes its task, the code cache becomes read-only again (access 4).

In this paper, we demonstrate the feasibility of such race-condition-based code cache injection attacks, through a proof-of-concept exploit against modern browsers that support the *Web Worker* [24] specification. Rather than relying on a permanently writable code cache [21], our attack leverages the race condition and can bypass the permission switching based W⊕X enforcement (Figure 1). In this attack, the malicious JS code utilizes web workers to create a multi-threaded environment. After forcing a worker thread into the compilation state, the main JS thread can exploit vulnerabilities of the browser to inject shellcode into the worker thread's code cache.

To fundamentally prevent such attacks, we propose secure dynamic code generation (SDCG), a new architecture that 1) enables dynamic code generation to comply with the W⊕X policy; 2) eliminates the race condition; 3) can be easily adopted; and 4) introduces more acceptable performance overhead compared with alternative solutions. SDCG achieves these goals through a multi-process-based architecture. Specifically, instead of generating and modifying the code in the same process as the generated code, SDCG relocates the DCG functionality to another trusted process. The code cache is built upon memory shared between the original process and the trusted process. In the original process, the code cache is mapped as RX; and in the trusted process, the same memory is mapped as WR. By doing so, the code cache remains read-only under all circumstances in the untrusted process, eliminating the race condition that allows the code cache to be writable to untrusted thread(s). At the same time, the code generator in the trusted process can freely perform code generation, patching and garbage collection as usual. To enable transparent interaction between the code generator and the generated code, we only need to add a few wrappers that make the code generator invocable through remote procedure calls (RPC). Since only functions that modify code cache need to be handled, the effort for adding wrappers is small.

We have implemented SDCG for two types of popular code generators: JS engine and DBT. For JS engine, our implementation is based on V8 [25]. For DBT, our implementation is based on Strata [9]. Our implementation experience showed that porting code generators to SDCG only requires a small modification: besides the shareable part, which is about 500 lines of C code (LoC), we only added about 2,500 LoC for V8 and about 1,000 LoC for Strata. We evaluated the security of SDCG and the performance overhead of our two prototype implementations. The results showed that SDCG are secure under our threat model and the performance overhead introduced by our prototype implementations is small: around 6.90% (32-bit) and 5.65% (64-bit) for V8 benchmark suite; and around 1.64% for SPEC CINT 2006 (additional to Strata's own overhead).

In summary, we made the following contributions:

- Beyond the known exploit technique against permanently writable code cache [21], we demonstrated the feasibility of exploiting race conditions to maliciously modify the code cache that is protected by permission switching based W⊕X enforcement; and discussed the severity of such attacks (Section III).

- We proposed secure dynamic code generation (SDCG), a multi-process-based architecture that provides better security (mandatory, non-bypassible W⊕X enforcement), low performance overhead, and easy adoption (Section IV).
- We implemented two prototypes of SDCG, one for V8 JS engine and one for Strata dynamic binary translator (Section V).
- We evaluated the performance overhead of our two prototype implementations (Section VI).

II. Related Work

In this section, we discuss the techniques that could be used to protect the code cache from being maliciously modified and explain their disadvantages. We also discuss other forms of attacks against the JIT engines and their countermeasures.

A. *Software-based Fault Isolation*

Software-based fault isolation (SFI) [26] can be used to confine a program's ability to access memory resources. On 32-bit x86 platforms, SFI implementations usually leverage segment registers [27], [28] to confine memory accesses, for the benefit of low runtime overhead. On other platforms without segment support (e.g., x86-64, ARM), SFI implementations use either address masking [29] or access control list (ACL) [30], introducing higher runtime overhead.

Once memory accesses — especially write accesses — are confined SFI can prevent untrusted code from overwriting security sensitive data, such as the code cache. Our SDCG solution differs from SFI in several respects. First, SFI's overhead comes from the execution of the extra inline checks; but SDCG's overhead comes from the remote procedure call and cache synchronization on multi-core systems. Therefore, if the execution mostly stays within the code cache, SDCG will introduce less overhead than SFI. On the other hand, if the execution needs to be frequently switched between the code generator and the generated code, then SFI could be faster. Since most modern code generators try to make the execution stay as long as possible in the code cache, our approach is more suitable in this scenario.

Second, to reduce the overhead of address masking, many SFI solutions [29] use ILP32 (32-bit integer, long, pointer) primitive data types, limiting data access to 4GB space, even on a 64-bit platform. On the other hand, SDCG does not have this limitation.

It is worth noting that, some efforts have been made to apply SFI to JIT engines [31], [32]. Despite relatively higher overhead, the threat model of these approaches usually did not consider scenarios where the JIT compiler is only a component of a larger software, such as a web browser. Since most vulnerabilities of web browsers are found outside the JIT engines [33], to apply such techniques, one would have to apply SFI to other browser components as well. This could result in even higher performance overhead. From this perspective, we argue that our solution is more realistic in practice.

B. *Memory Safety*

The attacks on the code caches (at randomized locations) rely on the ability to write to memory area specified by attacker. Therefore, such attacks could be defeated by memory safety enforcement, which prevents all unexpected memory read and write. However, many programs are written in low-level languages like C/C++ that are prone to memory corruption bugs, leading to a majority of security vulnerabilities. Unfortunately, existing memory safety solutions [34], [35], [36], [37], [38], [39], [40] for C/C++ programs tend to have much higher performance overhead than SFI or other solutions, prohibiting their adoptions. For example, the combination of Softbound [40] and CETS [39] provides a strong spatial and temporal memory safety guarantee, but they were reported to have 116% average overhead on SPEC CPU 2000 benchmark. Compared with this direction of research, even though SDCG provides less security guarantees, it is still valuable because it fully blocks a powerful attack vector with minimal runtime overhead.

C. *Control Flow Integrity*

Control flow hijacking is a key step in many real world attacks. As DEP becomes ubiquitous, more and more attacks rely on return-to-libc [3] or ROP [4] to hijack control flow. Many solutions [5], [6], [7] are thus proposed to enforce control flow integrity (CFI) policy. With the CFI policy, the program's control flow cannot be hijacked to unexpected locations. It can protect the code cache in some way, e.g., attackers cannot overwrite the code cache by jumping to arbitrary address of the code generator.

However, attackers can still utilize arbitrary memory write vulnerabilities to overwrite the code cache without breaking CFI. Once code cache is overwritten, the injected code could be invoked through normal function invocations, without breaking the static CFI policy.

Moreover, when extending CFI to dynamically generated code, without proper write protection, the embedded enforcement checks can also be removed once attackers can overwrite the code. From this perspective, SDCG is complementary to CFI because it guarantees one basic assumption of CFI: code integrity protection.

D. *Process Sandbox*

Delegation-based sandbox architecture, a.k.a. the broker model [41], has been widely adopted by the industry and used in Google Chrome [42], Windows 8 [43], Adobe Reader [44], and etc. In this architecture, the sandboxed process drops most of its privileges and delegates all security sensitive operations to the broker process. The broker process in turn, checks whether the request complies with the security policy. SDCG is also based on the same architecture. Using this architecture, we 1) delegate all the operations that will modify the code cache (e.g., code installation, patching and deletion) to the translator process; and 2) make sure the W⊕X policy is mandatory.

E. *Attacks on JIT engines*

Attacker have targeted the code cache for its writable and executable property. Currently, the most popular exploit technique is JIT spray [45], an extension to classic heap spray attacks [46]. Heap spray is used to bypass ASLR without guessing the address of injected shellcode. This technique became unreliable after DEP is deployed because the heap is no longer executable. To bypass this, attackers turned to JIT engines, The JIT spray attack abuses the JIT engine to emit chunks of predictable code, and then hijacks the control to the entry or middle of one of these code chunks. DEP or W⊕X is thus bypassed because these code chunks reside in the executable code cache. Most JIT engines have since deployed different mitigation techniques to make the layout of the code cache unpredictable, e.g., random NOP insertion, constant splitting, and etc. And researchers have also proposed more robust technique [47], [31] to prevent such attacks.

Rather than abusing the JIT engines to create expected code, attackers can also abuse the writable property of the code cache and directly overwrite the generated code [21]. In this paper, we first extend the attack [21] to show that, even with a permission switching based W⊕X enforcement, attackers can still leverage race conditions to bypass such enforcement. Then we propose a solution can fundamentally defeats all code-cache injection based attacks.

III. Attacking the Code Cache

In this section, we describe in detail the code cache injection threat we are addressing in this paper. We begin this section with our assumptions and threat model. Next, we show how code cache can be attacked to bypass the state-of-the-art exploit mitigation techniques. Finally, we demonstrate how a naive W⊕X enforcement can be bypassed by exploiting race conditions.

A. Assumptions and Threat Model

SDCG focuses on preventing remote attackers from leveraging the code cache as an attack vector to trigger arbitrary code execution. We focus on two classic attack scenarios discussed as follows. In both scenarios, we assume the code generator itself is trusted and does not have security vulnerabilities.

- *Foreign Attacks.* In this scenario, the code generator is a component of a program (e.g., a web browser). The program is benign, but components other than the code generator are assumed to be vulnerable when handling input or contents provided by attacker (e.g., a malicious web page). Attackers can then exploit the vulnerable components to attack the code cache.

- *Jail-break Attacks.* In this scenario, the code generator is used to sandbox or monitor an untrusted program, and attacks are launched within the code cache. This could happen under two circumstances. First, the sandboxed program itself is malicious. Second, the program is benign, but the dynamically generated code has vulnerabilities that can be exploited by attackers to jailbreak.

Without loss of generality, we assume that the following mitigation mechanisms for both general and JIT-based exploits have been deployed on the target system.

- *Address Space Layout Randomization.* We assume that the target system has deployed at least the base address randomization, and all the predictable memory mappings have been eliminated.

- *JIT Spray Mitigation.* For JIT engines, we assume that they have deployed a full-suite of JIT spray mitigation mechanisms, including but not limited to random NOP insertion, constant splitting and those proposed in [47], [31].

- *Guard Pages.* We assume the target system creates guard pages (i.e., pages without access permission) to wrap each pool of the code cache, like the Google V8 JS engine does. These guard pages can prevent buffer overflows, both overflows out of the code cache, and overflows into the code cache.

- *Page Permissions.* We assume that the underlying hardware has the support for mapping memory as non-executable (NX). And writable data memory like stack and normal heap are set to be non-executable. Furthermore, we assume that all the statically generated code has been set to non-writable to prevent overwriting. However, almost all JIT compilers map the code cache as both writable and executable.

The target system can further deploy the following advanced mitigation mechanisms for the purpose of sandboxing and monitoring:

- *Fine-grained Randomization.* The target system can enforce fine-grained randomization by permuting the order of functions [48] or basic blocks [49], randomizing location of each instruction [50], or even randomizing the instruction set [51], [52].

- *Control Flow Hijacking Mitigation.* The target system can deploy different kinds of control flow hijacking mitigation mechanisms, including (but not limited to): control flow integrity enforcement, either coarse-grained [7], [6] or fine-grained [5], [53]; return-oriented programming detection [54], [55]; and dynamic taint analysis based hijacking detection [17].

To allow overwriting of the code cache, we assume there is at least one vulnerability that allows attackers to write to attacker-specified address with attacker-provided contents. We believe this is a realistic assumption, because many types of vulnerabilities can be exploited to achieve this goal, such as format string [56], heap overflow [57], use-after-free [58], integer overflow [59], and and etc. For example, the attack described in [21] obtained this capability by exploiting an integer overflow vulnerability (CVE-2013-6632); and in [60], the author described how 5 use-after-free vulnerabilities (CVE-2013-0640, CVE-2013-0634, CVE-2013-3163, CVE-2013-1690, CVE-2013-1493) can be exploited to perform arbitrary memory write. It is worth noting that, in many attack scenarios, the ability to do arbitrary memory write can easily lead to arbitrary memory read and information disclosure abilities.

B. Overwriting the Code Cache

1) Software Dynamic Translator: For the ease of discussion, we use the term software dynamic translator (SDT) to represent software that leverages dynamic code generation to translate code in one format into another format. Before describing the attacks, we first give a brief introduction on SDT. A core task of all SDTs is to maintain a mapping between untranslated code and translated code. Whenever a SDT encounters a new execution unit (depending on the SDT, the execution unit could be a basic block, a function or a larger chunk of code), it first checks whether the execution unit has already been translated. If so, it switches to execute the already translated code residing in the code cache; otherwise, it translates this new execution unit and installs the translated code into the code cache.

2) Exploit Primitives: In this section, we describe how code cache with full `WRX` permission can be overwritten. This is done in two steps. First, we need to bypass ASLR and find out where the code cache locates. Second, we need to be able to write to the identified location.

a) Bypassing ASLR: The effectiveness of ASLR or any randomization based mitigation mechanism relies on two assumptions: i) the entropy is large enough to stop brute-force attacks; and ii) the adversary cannot learn the random value (e.g., module base, instruction set).

Unfortunately, these two assumptions rarely hold in practice. First, on 32-bit platforms, user space programs only have 8 bits of entropy for heap memory, which is subject to brute-force guessing [3] and spray attacks [46]. Second, with widely available information disclosure vulnerabilities, attackers can easily find out the random value [61], [62]. In fact, researchers have demonstrated that even with a single restricted information disclosure vulnerability, it is possible to traverse a large portion of the memory content [63].

In the scenario of attacking code cache, this status quo implies that we can either launch JIT spray attack to prepare a large number of `WRX` pages on platforms with low entropy; or leverage an information disclosure vulnerability to pinpoint exactly where the code cache is. Note that, as one only needs to know the location of the code cache, most fine-grained randomizations that try to further randomize the content of the memory are ineffective for this attack. Since the content of code cache will be overwritten in the next step (described below), none of the JIT spray mitigation mechanisms can provide effective protection against this attack.

b) Writing to Code Cache: Once obtaining the location of the code cache, the next step is to inject shellcode to the code cache. In most cases, the code cache will not be adjacent to other writable heap memory (due to ASLR), and may also be surrounded by guard pages. For these reasons, we cannot directly exploit a buffer overflow vulnerability to overwrite the code cache. However, as our assumption section suggests, besides logic errors that directly allow one to write to anywhere in the memory, several kinds of memory corruption vulnerabilities can also provide the arbitrary memory write ability. In the following example, an integer overflow vulnerability is exploited to acquire this capability.

3) An In-the-Wild Attack: We have observed one disclosed attack [21] that leveraged the code cache to achieve reliable arbitrary code execution. This attack targeted the mobile Chrome browser. By exploiting an integer overflow vulnerability, the attack first gained reliable arbitrary memory read and write capabilities. Using these two capabilities, the attack subsequently bypassed ASLR and located the permanently writable and executable code cache. Finally, it injected shellcode into the code cache and turned control flow to the shellcode.

4) Security Implication: In practice, we have only observed this single attack that injects code into the code cache. We believe this is mainly due to the convenience of a popular ROP attack pattern, which works as: i) preparing a traditional shellcode in memory; ii) exploiting vulnerabilities to launch ROP attack; iii) using the ROP gadgets to turn on the execution permission of the memory where the traditional shellcode resides; and iv) jumping to the traditional shellcode to finish the intended malicious tasks. However, once advanced control flow hijacking prevention mechanisms such as fine-grained CFI are deployed, this attack pattern will be much more difficult to launch.

On the contrary, the code cache injection attack can easily bypass most of the existing exploit mitigation mechanisms. First, all control flow hijacking detection/prevention mechanisms such as CFI and ROP detection rely on the assumption that the code cannot be modified. When this assumption is broken, these mitigation mechanisms are no longer effective. Second, any inline reference monitor based security solution is not effective because the injected code is not monitored.

C. Exploiting Race Condition

A *naive* defense against the code cache injection attack is to enforce W⊕X by manipulating page permissions (Figure 1). More specifically, when the code cache is about to be modified (e.g., for new code generation or runtime garbage collection), it turns on the write permission but turning off the execution permission ($t1$). And when the code cache is about to be executed, it turns off write permission but turn on execution permission ($t2$).

This solution prohibits the code cache to be both writable and executable at the same time. If the target program is single-threaded, this approach can prevent code cache injection attacks. Since the code cache is only writable when the SDT is executing and we assume that the SDT itself is trusted and not vulnerable, attackers cannot hijack or interrupt the SDT to overwrite the code cache. However, as illustrated in Figure 2, in more general multi-threaded programming environment, even if the SDT is trusted, the code cache can still be overwritten by other insecure threads (Thread B) when the the code cache is set to be writable for one thread (Thread A).

In this section, we use a concrete attack to demonstrate the feasibility of such attack, i.e., with the naive W⊕X enforcement, it is still possible to overwrite the code cache with the same exploit primitives described above.

1) Secure Page Permissions: Since the V8 JS engine does not have the expected page permission protection, i.e., the naive W⊕X enforcement, we implemented one in V8 for the demonstration of our attack.

Specifically, by default, when a memory region is allocated from the OS (e.g., via `mmap`) for the code cache, it is allocated as executable but not writable. We will turn on the write permission and turn off the execution permission of the code cache for:

- *New Code Installation.* Usually, the JavaScript program (e.g., a function) is first compiled into native code, and then copied into the code cache. To allow the copy operation, we need to turn on the write permission of the code cache.

- *Code Patching.* Existing code in the code cache is patched under certain circumstances. For instance, after a new code is copied into the code cache, its absolution address is thus determined; then instructions that require absolution address operands from this new code fragment is resolved and patched.

- *Runtime Inline Caching.* Inline caching is a special patching mechanism introduced to provide better performance for the JIT compiled programs written in dynamic typed languages. With runtime execution profile information, the JIT compiler caches/patches the result (e.g., the result of object property resolving) into the instructions in the code cache at runtime.

- *Runtime Garbage Collection.* The JavaScript engine needs to manage the target JavaScript program's memory, especially the garbage collection. The code cache needs to be modified for two main reasons. First, when unused code fragment needs to be removed from the code cache. Second, when a data object is moved to new addresses by the garbage collector, instructions referencing it have to be updated.

When these operations finish, or any code in the code cache needs to be invoked, we turn off the write permission of the code cache and turn on the execution permission.

To further reduce the attack surface, all the above policies are enforced in a fine-grained granularity. That is, 1) each permission change only covers memory pages that are accessed by the write or execution operations; and 2) the write permission is turned on only when an write operation is performed, and is turned off immediately after the write operation finishes. This fine-grained implementation provides a maximum protection for code caches.

2) Multi-threaded Programming in SDT: To launch the race-condition-based attack, we need two more programming primitives. First we need the ability to write multi-threaded programs. Note, some SDTs such as Adobe Flash Player also allows "multi-threaded" programming, but each "thread" is implemented as a standalone OS process. For these SDTs, since the code cache is only writable to the corresponding thread, our proposed exploit technique would not work. Second, since the attack window is generally small, we need the ability to coordinate threads before launching the attack.

- *Thread Primitives.* A majority of SDTs have multi-threaded programming support. JavaScript (JS) used to be single-threaded and event-driven. With the new HTML5 specification, JS also supports multi-threaded programming, through the specification of `WebWorker` [24]. There are two types of `WebWorker`: *dedicated* worker and *shared* worker. In V8, the dedicated worker is implemented as a thread within the same process; and shared worker is implemented as a thread in a separated process. Since we want to attack one JS thread's code cache with another JS thread, we leverage the dedicated worker. Note that although each worker thread has its own code cache, it is still possible to launch the attack, because memory access permissions are shared by all threads in the same process.

- *Synchronization Primitives.* To exploit the race condition, two attacker-controlled threads need to synchronize their operations so that the overwritten can happen within the exact time window when the code cache is writable. Since synchronization is an essential part of multi-threaded programming, almost all SDTs support thread synchronization. In JS, thread synchronization uses the `postMessage` function.

3) A Proof-of-Concept Attack: Based on the vulnerability disclosed in the previous real-world exploit, we built a proof-of-concept race-condition-based attack on the Chrome browser. Since the disclosed attack [21] already demonstrated how ASLR can be bypassed and how arbitrary memory write capability can be acquired, our attack focuses on how race conditions can be exploited to bypass naive W⊕X enforcement. The high level workflow of our attack is as follows:

i) *Creating a Worker.* The main JS thread creates a web worker, and thus a worker thread is created.

ii) *Initialize the Worker.* The worker thread initializes its environment, making sure the code cache is created. And then, it sends a message to the main thread through `postMessage` to inform that it is ready.

iii) *Locating the Worker's Code Cache.* Upon receiving the worker's message, the main JS thread locates the worker thread's code cache, e.g., by exploiting an information disclosure vulnerability. In Chrome V8 engine, attackers can locate the code cache like the previous disclosed exploit. But instead of following the pointers for the current thread, attackers should go through the thread list the JS engine maintains and follow pointers for the worker thread. Then, it informs the worker that it is ready too.

iv) *Making Code Cache Writable.* Upon receiving the main thread's message, the worker thread begins to execute another piece of code, forcing the SDT to update its code cache. In V8, the worker can execute a function that is big enough, forcing the SDT to create a new `MemoryChunk` as the code fragment and set it to be writable (for a short time).

v) *Monitor and Overwrite the Code Cache.* At the same time, the main thread keeps monitoring the status of the code cache, and tries to overwrite the code cache once its status is updated. In V8, the main thread can keep polling the head of `MemoryChunk` linked list to identify the creation of a new code fragment. Once a new code fragment is created, the main thread can then monitor its content. Once the first few bytes (e.g., the function prolog) are updated, the main thread can try to overwrite the code cache to inject shellcode. After overwriting, the main thread informs the worker it has finished.

vi) *Executing the Shellcode.* Upon receiving the main thread's new message, the worker calls the function whose content has already been overwritten. In this way, the injected shellcode gets executed.

It is worth noting that, the roles of the main thread and the worker thread cannot be swapped in practice. The reason is, worker threads do not have access to the document object model (DOM). Since lots of vulnerabilities are within the rendering engine instead of the JS engine, this means only the main thread who has the access to the DOM can exploit those vulnerabilities.

4) Reliability of Race Condition: One important question for any race-condition-based attack is its reliability. The first factor that can affect the reliability of our attack is synchronization, i.e., the synchronization primitive should be fast enough so that the two threads can carry out the attack within the relatively small attack window. To measure the speed of the synchronization between the worker and the main thread, we ran another simple experiment:

i) The main thread creates a worker thread;
ii) The worker thread gets a timestamp and sends it to the main thread;
iii) Upon receiving the message, the main thread sends an echo to the worker;
iv) Upon receiving the message, the worker thread sends back an echo;
v) The main thread and the worker repeatedly send echoes to each other in this way for 1,000 times.
vi) The main thread gets another timestamp and computes the time difference.

The result shows that the average synchronization delay is around 23 μs. Comparing with this, the average attack window ($t2-t1$ in Figure 2) of our fine-grained naive W$\oplus$X protection is about 43 μs. Thus in theory, the `postMessage` method is sufficiently fast to launch a race condition attack.

The second and more important factor that can affect the reliability of our attack is task scheduling. Specifically, if the thread under the SDT context (e.g., the worker thread) is de-scheduled by the OS while the attacking thread (e.g., the main thread) keeps executing, then the attacking window will be enlarged. The bad news (for defenders) is that, the only way to change the code cache's memory permission is through a system call, and a context switch is likely to happen during the system call. For example, the system call for changing memory access permission on Linux is `mprotect`. During the invocation of `mprotect`, since we are using fine-grained protection, the virtual memory area needs to be split or merged, it will trigger the thread to be de-scheduled. As a result, the main thread (with higher priority than the worker) can gain control to launch attacks.

Combining these two factors, our demo's result shows that the race-condition-based attack can succeed with a very high probability. We have tested the browser for 100 times, and the attack succeeded for 91 times.

IV. System Design

In this section, we present the design of `SDCG`. We have two design goals: 1) `SDCG` should prevent all possible code injection attacks against the code cache under our adversary model; and 2) `SDCG` should only introduce trivial performance overhead. In addition, `SDCG` is designed to be integrated with the targeted SDT, and we assume that the source code of the SDT is available.

A. Overview and Challenges

Since the root cause of the attack is writable code cache (either permanently or temporarily), we can be prevent such attacks by two design choices: 1) ensuring that no one but the SDT can write to the code cache, e.g., through SFI or memory safety; and 2) ensuring that the memory occupied by the code cache is always mapped as `RX`. We chose the second choice for two major reasons. First, we expect that the performance overhead of applying SFI or memory safety to an entire, large, complex program (e.g., the web browser) would be very high. Second, implementing the first choice requires significant engineering effort.

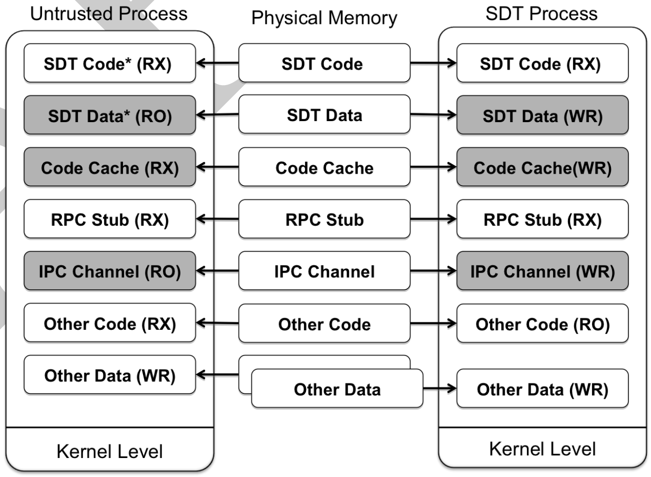

Fig. 3: Overview of `SDCG`' multi-process-based architecture. The gray memory areas are shared memory, other are mapped as private (copy-on-write). Depending on the requirement, the SDT's code and data can be mapped differently.

Figure 3 shows the high level design of `SDCG`. The key idea is that, through shared memory, the same memory content will be mapped into two (or more) different processes, with different access permissions. In the untrusted process(es), the code cache will be mapped as `RX`; but in the SDT process, it will be mapped as `WR`. By doing so, `SDCG` prevents any untrusted code from modifying the code cache. At the same time, it allows the SDT to modify the code cache as usual. The only difference is, whenever the SDT needs to be invoked, e.g., to install a new code fragment, the request will be served through remote procedure call (RPC) instead of a normal function call.

To build and maintain this memory model, we need to solve following technical and engineering challenges.

i) *Memory Map Synchronization*. Since the memory regions occupied by the code cache are dynamically allocated and can grow and shrink freely, we need an effective way to dynamically synchronize memory mapping between the untrusted process(es) and the SDT process. More importantly, to make `SDCG`'s protection mechanism work transparently, we have to make sure that the memory is mapped at exactly the same virtual address in all processes.

ii) *Remote Procedure Call*. After relocating the SDT to another process, we need to make it remotely invocable, by wrapping former local invocations with RPC stubs. Since RPC is expensive, we need to reduce the frequency of invocations, which also reduces the attack surface.

iii) *Permission Enforcement*. Since `SDCG`'s protection is based on memory access permissions, we must make sure that untrusted code cannot tamper with our permission scheme. Specifically, no memory content can be mapped as both writable and executable, neither at the same time nor alternately.

B. Memory Map Synchronization

Synchronizing memory mapping between the untrusted process(es) and the SDT process is a bi-directional issue. On the one hand, when the SDT allocates a new code fragment in the SDT process, we should map the same memory region in the untrusted process(es) at exactly the same address; otherwise the translated code will not work correctly (e.g., creating incorrect branching target). On the other hand, the untrusted process may also allocate some resources that are critical to the SDT. For example, in the scenario of binary translation, when the untrusted process loads a dynamically linked module, we should also load the same module at the same address in the SDT process; otherwise the SDT will not be able to locate the correct code to be translated. Moreover, we want this synchronization to be as transparent to the SDT as possible, so we can keep the changes minimal.

When creating the shared memory, there are two possible strategies: on-demand and reservation-based. On-demand mapping creates the shared memory at the very moment a new memory region is required, e.g., when the SDT wants to add a new memory region to the code cache. However, as the process address space is shared by all modules of a program, the expected address may not always be available in both the untrusted process and the SDT process. For this reason, we choose the reservation-based strategy. That is, when the process is initialized, we reserve (map) a large chunk of shared memory in both the untrusted process(es) and the SDT process. Later, any request for shared memory will be allocated from this shared memory pool. Note that, in modern operation systems, physical memory resources are not mapped until the reserved memory is really accessed, so our reservation-based strategy does not impose significant memory overhead.

Once the shared memory pool is created, synchronization can be done via inter-process communication (IPC). Specifically, when the SDT allocates a new memory region for the code cache, it informs the untrusted process(es) about the base address and the size of this new memory region. Having received this event, the untrusted process(es) maps a memory region with the same size at the same base address with the expected permission (`RX`). Similarly, whenever the untrusted process allocates memory that needs to be shared, a synchronization event is sent to the SDT process.

C. Remote Procedure Call

Writing RPC stubs for the SDT faces two problems: argument passing and performance. Argument passing can be problematic because of pointers. If a pointer points to a memory that is different between the untrusted process and the SDT process, then the SDT ends up using incorrect data and causes run-time errors. Vise versa, if the returned value from the SDT process contains pointers that point to data not copied back, the untrusted code ends up running incorrectly. One possible solution is that, instead of passing the pointer to the remote process, the stub serializes the object before passing it to the remote process. Unfortunately, not all arguments have built-in serialization functionality. In addition, when an argument is a large object, performing serialization and copy for every RPC invocation introduces high performance overhead. Thus, in general, stub generation is not easy without the support from the SDT or support from program analysis.

To avoid this problem, `SDCG` takes a more systematic approach. Specifically, based on the observation that a majority of data that the SDT depends on is either read-only, or resides in dynamically mapped memory, we extend the shared memory to also include the dynamic data the SDT depends on. According to the required security guarantee, the data should be mapped with different permissions. By default, `SDCG` maps the SDT's dynamic data as read-only in the untrusted process, to prevent tamper from the untrusted code. However, if non-control data attacks are not considered, the SDT's dynamic data can be mapped as `WR` in the untrusted process. After sharing the data, we only need to handle a few cases where writable data (e.g., pointers within global variables) is not shared/synchronized.

Since RPC invocations are much more expensive than normal function calls, we want to minimize the frequency of RPC invocation. To do so, we take a passive approach. That is, we do not convert an entry to the SDT to RPC unless it modifies the code cache. Again, we try to achieve this goal without involving heavy program analysis. Instead, we leverage the regression tests that are usually distributed along with the source code. More specifically, we begin with no entries being converted to RPC and gradually convert them until all the regression tests can pass.

While our approach can be improved with more automation and program analysis, we leave these as future work because our main goal here is to design and validate that our solution is effective against the new cod cache injection attacks.

D. Permission Enforcement

To enforce mandatory W$\oplus$X, we leverage the delegation-based sandbox architecture [41]. Specifically, we intercept all system calls related to virtual memory management, and enforce the following policies in the SDT process:

(I) No memory can be mapped as both writable and executable.
(II) When mapping a memory region as executable, the base address and the size must come from the SDT process, and the memory is always mapped as RX.
(III) The permission of non-writable memory cannot be changed.

E. Security Analysis

In this section, we analyze the security of SDCG under our threat model. First, we show that our design can enforce permanent W⊕X policy. The first system call filtering policy ensures that attackers cannot map memory that is both writable and executable. The second policy ensures that attackers cannot switch memory from non-executable to executable. The combination of these two policies guarantees that no memory content can be mapped as both writable and executable, neither at the same time nor alternately. Next, the last policy ensures that if there is critical data that the SDT depends on, it cannot be modified by attackers. Finally, since the SDT is trusted and its data is protected, the second policy can further ensure that only SDT-verified content (e.g., code generated by the SDT) can be executable. As a result, SDCG can prevent any code injection attack.

V. Implementation

We implemented two prototypes of SDCG, one for Google V8 JS engine [25], and the other for Strata DBT [9]. Both prototypes were implemented on Linux. We chose these two SDTs for the following reasons. First, JS engine is one of the most widely deployed SDT. At the same time, it is also one of the most popular stepping stone for launching attacks. Among all JS engines, we chose V8 because it is open sourced, highly ranked, and there is disclosed exploit [21]. Second, DBT have been widely used by security researchers to build various security solutions [16], [17], [18], [19], [20]. Among all the DBTs, we chose Strata because 1) it has been used to implement many promising security mechanisms, such as instruction set randomization [19], instruction layout randomization [20], etc.; and 2) its academic background allowed us to have access to its source code, which is required for implemented SDCG.

A. Shared Infrastructure

The memory synchronization mechanism and the system call filtering mechanism are specific to the target platform; but they can be shared among all SDTs.

1) Seccomp-Sandbox: Our delegation-based sandbox is built upon the seccomp-sandbox [64] from Google Chrome. Although Google Chrome has switched to a less complicated process sandbox based on seccomp-bpf [65], we found that the architecture of seccomp-sandbox serves our goal better. Specifically, since seccomp [66] only allows four system calls once enabled, and not all system calls can be fulfilled by the broker (e.g., mmap), the seccomp-sandbox introduced a trusted thread to perform system calls that cannot be delegated to the broker. To prevent attacks on the trusted thread, the trusted thread operates entirely on CPU registers and does not trust any memory that is writable to the untrusted code. When the trusted thread makes a system call, the system call parameters are first verified by the broker, and then passed through a shared memory that is mapped as read-only in the untrusted process. As a result, even if the other threads in the same process are compromised, they cannot affect the execution of the trusted thread. This provides us a perfect foundation to securely build our memory synchronization mechanism and system call filtering mechanism.

To enforce the mandatory W⊕X policy, we modified the sandbox so that, before entering the sandbox mode, SDCG enumerates all memory regions and converts any WRX region to RX.

For RPC invocation, we also reused seccomp-sandbox's domain socket based communication channel. However, we did not leverage the seccomp mode in our current implementation. There are several reasons. First, it is not compatible with the new seccomp-bpf-based sandbox used in Google Chrome. Second, it intercepts too many system calls that are not required by SDCG. More importantly, both Strata and seccomp-bpf provide enough capability for system call filtering.

2) Shared Memory Pool: During initialization, SDCG reserves a large amount of consecutive memory as a pool. This pool is mapped as shared (MAP_SHARED), not file backed (MAP_ANONYMOUS) and with no permission (PROT_NONE). After this, any mmap request from the SDT allocates memory from this pool (by changing the access permission), instead of going to the mmap system call. This guarantees any SDT allocated region can be mapped at exactly the same address in both the SDT process and the untrusted process(es).

After the sandbox is enabled, whenever the SDT calls mmap, SDCG generates a synchronized request to the untrusted process(es), and wait until the synchronization is done before returning to the SDT. In the untrusted process, the synchronization event is handled by the trusted thread. It reads synchronization request from the IPC channel and then changes the access permission of the given region to the given value. Since the parameters (base address, size and permission) are passed through the read-only IPC channel and the trusted thread does not use stack, it satisfies our security policy for mapping executable memory.

Memory mapping in the untrusted process(es) is forwarded to the SDT process by the system call interception mechanism of the sandbox. The request first goes through the system call filtering to make sure the security policy in enforced. SDCG then checks where the request comes from. If the request is from the SDT, or is a special resource the SDT depends on (e.g., mapping new modules needs to be synchronized for Strata), the request is fulfilled from the shared memory pool. If it is a legitimate request from the untrusted code, the request is fulfilled normally.

3) System Call Filtering: SDCG rejects the following types of system calls.

- mmap with writable (PROT_WRITE) and executable (PROT_EXEC) permission.
- mprotect or mremap with target region falls into a protected memory region.
- mprotect with executable (PROT_EXEC) permission.

`SDCG` maintains a list of protected memory regions. After the SDT process is forked, it enumerates the memory mapping list through `/proc/self/maps`, and any region that is executable is included in the list. During runtime, when a new executable region is created, it is added to the list; and when a region is unmapped, it is removed from the list. If necessary, the SDT's dynamic data can also be added to this list.

For Strata, this filtering is implemented by intercepting the related system calls (`mmap`, `mremap` and `mprotect`). For V8 (integrated with the Google Chrome browser), we rely on the seccomp-bpf filtering policies.

B. SDT Specific Handling

Next, we describe some implementation details that are specific to the target SDT.

1) Implementation for Strata: Besides the code cache, many Strata-based security mechanisms also involve some critical metadata (e.g., the key to decrypt randomized instruction set) that needs to be protected. Otherwise, attackers can compromise such data to disable or mislead critical functionalities of the security mechanisms. Thus, we extended the protection to Strata's code, data, and the binary to be translated. Fortunately, since Strata directly allocates memory from `mmap` and manages its own heap, this additional protection can be easily supported by `SDCG`. Specifically, `SDCG` ensures that all the memory regions allocated by Strata is mapped as either read-only or inaccessible. Note that, we do not need to protect Strata's static data, because once the SDT process is forked, the static data is copy-on-write protected, i.e., while the untrusted code could modify Strata's static data, the modification cannot affect the copy in the SDT process.

Writing RPC stubs for Strata also reflects the differences in attack model: since all dynamic data are mapped as read-only, any functionality that modified the data also needs to be handled in the SDT process.

Another special case for Strata is the handling of process creation, i.e., the `clone` system call. The seccomp-sandbox only handles the case for thread creation, which is sufficient for Google Chrome (and V8). But for Strata, we also need to handle process creation. The challenge for process creation is that, once a memory region is mapped as shared, the newly created child process will also inherit this memory regions as shared. Thus, once the untrusted code forks a new process, this process also shares the same memory pool, with its parent and the SDT process. If we want to enforce an $1:1$ serving model, then we need to un-share the memory. Unfortunately, un-sharing memory under Linux is not easy: one needs to 1) map a temporary memory region, 2) copy the shared content to this temporary region, 3) unmap the original shared memory, 4) map a new shared memory region at exactly the same address, 5) copy the content back, and 6) unmap the temporary memory region. At the same time, the child process is very likely to either share the same binary as its parent, which means it can be served by the same SDT; or call `execve` immediately after the fork, which completely destroys the virtual address space it inherited from its parent. For these reasons, we implemented a $N:1$ serving model for Strata, i.e., one SDT process serves multiple untrusted processes. And the `clone` system call can then be handled in the same way for both thread creation and process creation. The only difference is that, when a new memory region is allocated from the shared memory pool, all processes need to be synchronized.

2) Implementation for V8: Compared with Strata, the biggest challenge for porting V8 to `SDCG` is the dynamic data used by V8. Specifically, V8 has on two types of dynamic data: JS related data, and its own internal data. The first type of data is allocated from custom heaps that are managed by V8 itself. Similar to Strata's heap, these heaps directly allocate memory from `mmap`, thus `SDCG` can easily handle this type of data. The difficulty is from the second type of data, which is allocated from the standard C library (glibc on Linux). This makes it very challenging to track which memory region is used by the JS engine, and which is not. Clearly, we cannot make the standard C library to allocate all the memory from the shared memory pool. However, as mentioned earlier in the design section, we have to share data that is involved in the RPC so as to avoid serializing objects, especially C++ objects, which can be very complicated. To solve this problem, in our prototype implementation, we implemented a simple arena-based heap that is backed by the shared memory pool; and modified V8 to allocate certain objects from this heap. That is, only objects that are involved in the RPC need to be allocated from this heap, the rest can still be allocated from the standard C library.

Another problem is stack. Strata does not share the same stack as the translated program, so it never reads data from the program's stack. This is not true for V8. In fact, many objects used by V8 are allocated on the stack. Thus, during RPC handling, the STD process may dereference pointers pointing to the stack. Moreover, since the stack is assigned during thread creation, it is very difficult to enforce that the program always allocates stack from our shared memory pool. As a result, we ended up copying stack content between the two processes. Fortunately, only 3 RPCs require stack copy. Note that, because the content is copied to/from the same address, when creating the trusted SDT process, we must assign it a new stack, instead of relying on copy-on-write.

Writing RPC stubs for V8 is more flexible than Strata because the dynamic data is not protected. For this reason, we would prefer to convert functions that are invoked less frequently. To achieve this goal, we followed two general principles. First, between the entry of the JS engine and the point where the code cache is modified, many functions could be invoked. If we convert a function too high in the calling chain, and the function does not result in modification of the code cache under other context, we end up introducing unnecessary RPC overhead. For instance, the first time a regular expression is evaluated, it is compiled; but the next time, the compiled code can be retrieved from the cache. Thus, we want to convert functions that are post-dominated by operations that modify the code cache. On the other hand, if we convert a function that is too low in the calling chain, even though the invocation of this function always result in modification of the code cache, the function may be called from a loop, e.g., marking process during garbage collection. This also introduces unnecessary overhead. Thus, the second principle is that we want to convert functions that dominate as many modifications as possible. In our prototype implementation, since we did not use program analysis, these principles were followed empirically. In the end, we added a total of 20 RPC stubs.

VI. Evaluation

In this section, we describe the evaluation of the effectiveness and performance overhead of our two prototype implementations.

A. Setup

For our port of the Strata DBT, we measured the performance overhead using SPEC CINT 2006 [67]. Our port of the V8 JS engine was based on revision 16619. And the benchmark we used to measure the performance overhead is the V8 Benchmark distributed with the source code (version 7) [68]. All experiments were run on a workstation with one Intel Core i7-3930K CPU (6-core, 12-thread) and 32GB memory. The operating system is the 64-bit Ubuntu 13.04 with kernel 3.8.0-35-generic.

B. Effectiveness

In Section IV-E, we provided a security analysis of our system design, which shows that if implemented correctly, SDCG can prevent all code cache injection attacks. In this section, we evaluated our SDCG-ported V8 prototype to see if it can truly prevent the attack we demonstrated in Section III-C3.

The experiment was done using the same proof-of-concept code as described in Section III-C3. As the attack relies on race condition, we executed it for 100 times. For the version that is protected by naive W$\oplus$X enforcement, the attack was able to inject shellcode into the code cache for 91 times. But for SDCG-ported version, all 100 tries failed.

C. Micro Benchmark

The overhead introduced by SDCG comes from two major sources: RPC invocation and cache coherency.

1) RPC Overhead: To measure the overhead for each RPC invocation, we inserted a new field in the request header to indicate when this request is sent. Upon receiving the request, the handler then calculates the time elapsed between this and the current time. Similarly, we also calculated the time elapsed between the sent and the receiving of return values. To eliminate the impact from cache synchronization, we pinned all threads (in both the untrusted process and the SDT process) to one single core.

Besides the overhead for each RPC invocation, another important fact that affects the overall overhead is the frequency of RPC invocation, i.e., the more frequent an RPC is invoked, the worse the performance is. So we also collected this number during the evaluation.

Table I shows the result from the V8 benchmark, using the 64bit release build. The average latency for call request is around 3-4 μs and the average latency for RPC return is around 4-5 μs. So the average latency for an RPC invocation through SDCG's communication channel is around 8-9 μs. The number of RPC invocations is between 1,525 and 6,000. Since the input is fixed, this number is stable, with a small fluctuation caused by garbage collection. Comparing with the overall overhead presented in the next section, it meets the expectation that the larger the number of invocations is, the higher the overhead is. Among all RPC invocations, less than 24% require stack copy.

2) Cache Coherency Overhead: SDCG involves at least three concurrently running threads: the main thread in the untrusted process, the trusted thread in the untrusted process, and the main thread in the SDT process. This number can be more if the SDT to be protected already uses multiple threads. On a platform with multiple cores, all these threads can be scheduled to different cores. Since SDCG heavily depends on shared memory, the OS scheduling for these threads can also affect the performance, i.e., cache synchronization between threads executing on different cores introduces additional overhead.

In this section, we report this overhead at the RPC invocation level. In next section, we will present its impact on the overall performance. The evaluation also used V8 benchmark. To reduce the possible combination of scheduling, we disabled all other threads in V8, leaving only the aforementioned three threads. The Intel Core i7-3930K CPU on our test bed has 6 cores. Each core has a dedicated 32KB L1 data cache and 256KB integrated L2 cache. A 12MB L3 cache is shared among all cores. When Hyperthreading is enabled, each core can execute two concurrent threads.

Given the above configuration, we have tested the following scheduling:

1) All threads on single CPU thread (affinity mask = {0});
2) All threads on single core (affinity mask = {0,1});
3) Two main threads that frequently access shared memory on single CPU thread, trusted thread freely scheduled (affinity mask = {0},{*});
4) Two main threads on single core, trusted thread freely scheduled (affinity mask = {0,1},{*});
5) All three threads on different core (affinity mask = {0},{2},{4}); and
6) All three threads freely scheduled (affinity mask = {*},{*},{*}).

Table II shows the result, using 64bit release build. All the numbers are for RPC invocation, the return latency is omitted. Based on the result, it is clear that the scheduling has a great impact on the RPC latency. If the two main threads are not scheduled on the same CPU thread, the average latency can exacerbate to 3x-4x slower. On the other hand, the scheduling for trusted thread has little impact on the RPC latency. This is expected, because the trusted thread is only waken up for memory synchronization.

D. Macro Benchmark

In this section, we report the overall overhead SDCG introduces. Since OS scheduling can have a large impact on performance, for each benchmark suite, we evaluated two CPU schedules. The first one (*Pinned*) pins both main threads from the untrusted process and the SDT process to one single core; and the second one (*Free*) allows the OS to freely schedule all threads.

1) SPEC CINT 2006: Both the vanilla Strata and the SDCG-ported Strata are built as 32-bit. The SPEC CINT 2006 benchmark suite is also compiled as 32-bit. Since all benchmarks from the suite are single-threaded, the results of

TABLE I: RPC Overhead During the Execution of V8 Benchmark.

	Avg Call Latency	Avg Return Latency	# of Invocations	Stack Copy (%)	No Stack Copy (%)
Richards	4.70 μs	4.54 μs	1525	362 (23.74%)	1163 (76.26%)
DeltaBlue	4.28 μs	4.46 μs	2812	496 (17.64%)	2316 (82.36%)
Crypto	3.99 μs	4.28 μs	4596	609 (13.25%)	3987 (86.75%)
RayTrace	3.98 μs	4.00 μs	3534	715 (20.23%)	2819 (79.77%)
EarlyBoyer	3.87 μs	4.28 μs	5268	489 (9.28%)	4779 (90.72%)
RegExp	3.82 μs	5.06 μs	6000	193 (3.22%)	5807 (96.78%)
Splay	4.63 μs	5.04 μs	5337	1187 (22.24%)	5150 (77.76%)
NavierStokes	4.67 μs	4.82 μs	1635	251 (15.35%)	1384 (84.65%)

TABLE II: Cache Coherency Overhead Under Different Scheduling Strategies.

	Schedule 1	Schedule 2	Schedule 3	Schedule 4	Schedule 5	Schedule 6
Richards	4.70 μs	13.76 μs	4.47 μs	14.25 μs	12.85 μs	13.37 μs
DeltaBlue	4.28 μs	13.29 μs	4.31 μs	13.85 μs	14.09 μs	15.84 μs
Crypto	3.99 μs	10.91 μs	3.98 μs	14.07 μs	12.47 μs	13.48 μs
RayTrace	3.98 μs	14.99 μs	4.05 μs	14.76 μs	13.15 μs	12.35 μs
EarlyBoyer	3.87 μs	13.70 μs	3.87 μs	14.27 μs	13.42 μs	13.47 μs
RegExp	3.82 μs	14.64 μs	3.85 μs	14.48 μs	13.55 μs	12.32 μs
Splay	4.63 μs	12.92 μs	4.49 μs	13.22 μs	13.36 μs	15.11 μs
NavierStokes	4.67 μs	12.06 μs	4.47 μs	13.02 μs	14.80 μs	12.65 μs

different scheduling strategies only reflect the overhead caused by SDCG.

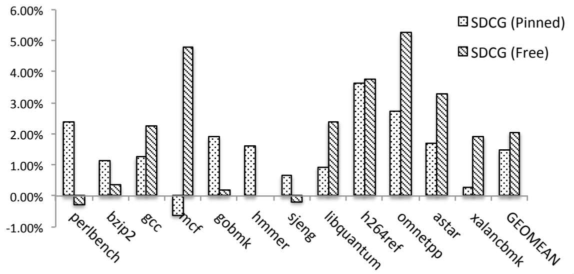

Fig. 4: SPEC CINT 2006 Slowdown. The baseline is the vanilla Strata.

	Native	**Strata**	**SDCG (Pinned)**	**SDCG (Free)**
perlbench	364	559	574	558
bzip2	580	600	613	602
gcc	310	403	420	410
mcf	438	450	479	471
gobmk	483	610	623	611
hmmer	797	777	790	777
sjeng	576	768	784	767
libquantum	460	463	511	474
h264ref	691	945	980	971
omnetpp	343	410	450	428
astar	514	546	587	563
xalancbmk	262	499	515	504
GEOMEAN	461	566	592	576

TABLE III: SPEC CINT 2006 Results. Since the standard deviation is quite small (less than 1%), we omitted this information.

Table III shows the evaluation result. The first column is the result of running *natively*. The second column is the result for Strata, but without SDCG. We use this as the baseline for calculating the slowdown introduced by SDCG. The third column is the result for SDCG with pinned schedule, and the last column is the result for SDCG with free schedule. Since the standard deviation is quite low (less than 1%), we omitted this information.

The corresponding slowdown is shown in Figure 4. For all benchmarks, the slowdown introduced by SDCG is less than 6%. And the overall (geometric mean) slowdown is 1.46% for pinned schedule, and 2.05% for free schedule.

Since SPEC CINT is a computation-oriented benchmark suite and Strata does a good job reducing the number of translator invocations, we did not observe a big difference between pinned schedule and free schedule.

2) JavaScript Benchmarks: Our port of V8 JS engine was based on revision 16619. For better comparison with SFI-based solution [31], we performed the evaluation on both IA32 and x64 release builds. The arena-based heap we implemented was only enabled for SDCG-ported V8. And to reduce the possible combination of scheduling, we also disabled all other threads in V8.

Table IV shows the results for IA32 build, and Table V shows the results for x64 build. The first column is the baseline result; the second column is the result of SDCG-ported V8 with pinned schedule; and the last column is the result of SDCG-ported V8 with free schedule. All results are the geometric mean over 10 executions of the benchmark. The number in the parentheses is the standard deviation in percentage. As we can see, the fluctuation is small; with the baseline and free schedule slightly higher than pinned schedule.

The corresponding slowdown is shown in Figure 5 (for IA32 build) and Figure 6 (for x64 build). Overall, we did not observe a big different between IA32 build and x64 build. For four benchmarks (Richards, DeltaBlue Crypto, NavierStokes), the slowdown introduced by SDCG is less than 5%, which is negligible because they are similar to the standard deviation.

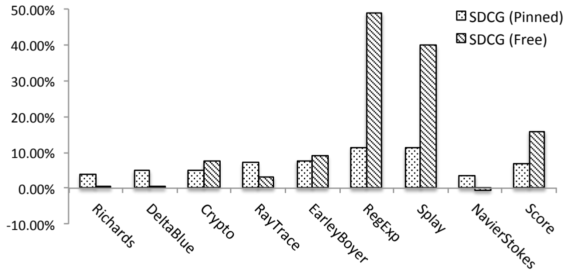

Fig. 5: V8 Benchmark Slowdown (IA32).

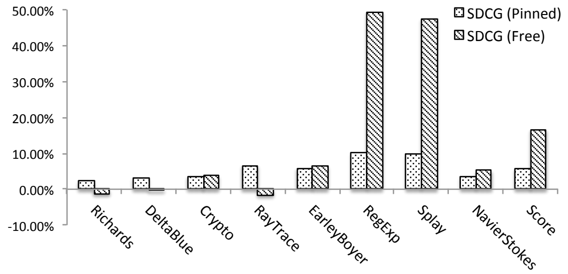

Fig. 6: V8 Benchmark Slowdown (x64).

	Baseline	**SDCG (Pinned)**	**SDCG (Free)**
Richards	24913 (2.76%)	23990 (0.28%)	24803 (1.72%)
DeltaBlue	25657 (3.31%)	24373 (0.43%)	25543 (3.86%)
Crypto	20546 (1.61%)	19509 (1.27%)	19021 (1.95%)
RayTrace	45399 (0.38%)	42162 (0.75%)	43995 (6.46%)
EarlyBoyer	37711 (0.61%)	34805 (0.27%)	34284 (0.82%)
RegExp	4802 (0.34%)	4251 (1.04%)	2451 (3.82%)
Splay	15391 (4.47%)	13643 (0.71%)	9259 (8.18%)
NavierStokes	23377 (4.15%)	22586 (0.42%)	23518 (1.26%)
Score	21071 (0.72%)	19616 (0.35%)	17715 (1.86%)

TABLE IV: V8 Benchmark Results (IA32). The score is the geometric mean over 10 executions of the benchmark suite. Number in the parentheses is the standard deviation.

	Baseline	**SDCG (Pinned)**	**SDCG (Free)**
Richards	25178 (3.39%)	24587 (2.31%)	25500 (3.24%)
DeltaBlue	24324 (3.65%)	23542 (0.38%)	24385 (2.54%)
Crypto	21313 (3.16%)	20551 (0.26%)	20483 (2.57%)
RayTrace	35298 (5.97%)	32972 (1.03%)	35878 (1.66%)
EarlyBoyer	32264 (4.42%)	30382 (0.61%)	30135 (1.04%)
RegExp	4853 (3.59%)	4366 (0.82%)	2456 (7.72%)
Splay	13957 (6.02%)	12601 (2.92%)	7332 (9.85%)
NavierStokes	22646 (2.48%)	21844 (0.30%)	21468 (3.45%)
Score	19712 (3.57%)	18599 (0.62%)	16435 (1.03%)

TABLE V: V8 Benchmark Slowdown (x64). The score is the geometric mean over 10 executions of the benchmark suite. Number in the parentheses is the standard deviation.

The rest four benchmarks (RayTrace, EarlyBoyer, RegExp, Splay) have higher overhead, but with pinned schedule, the slowdown is within 11%, which is much smaller than previous SFI-based solution [31] (79% on IA32).

There are the two major overhead sources. For RPC overhead, we can see a clear trend that the more RPC invocation is (Table I), the larger the slowdown is. However, the impact of cache coherency overhead caused by different scheduling strategies is not that consistent. For some benchmarks (Richards, DeltaBlu, RayTrace), the free scheduling is faster than the pinned scheduling. For some benchmarks (Crypto, EarlyBoyer), the overhead is almost the same. But for two benchmarks (RegExp and Splay), the overhead under free scheduling is much higher than the pinned scheduling. We believe this is because these two benchmarks depend more heavily on data (memory) access. Note that, unlike Strata, for SDCG-ported V8, we not only shared the code cache, but also shared the heaps used to store JS objects, for the ease of RPC implementation. Besides RPC frequency, this is another reason why we observed a higher overhead compared with SDCG-ported Strata.

VII. Discussion

In this section, we discuss the limitations of this work and potential future work.

A. Reliability of Race Condition

Although we only showed the feasibility of the attack in one scenario, in practice, the dynamic translator can be invoked under different situations, each of which has its own race condition window. Some operations can be very quick (e.g., patching), others may take a longer time to finish. By carefully controlling how the translator is invoked, we can enlarge the race condition window and make such attack more reliable.

In addition, OS scheduling can also affect the size of the attack window. For example, as we have discussed in the Section III, the invocation of `mprotect` is very likely to cause the thread to be swapped out of the CPU, which will enlarge the attack window.

B. RPC Stub Generation

To port a dynamic translator to SDCG, our current solution is to manually rewrite the source code. Even though the modification is relatively small compared to the translator's code size, the process still requires the developer to have a good understanding of the internals of the translator. This process can be improved or even automated through program analysis. Firstly, our current RPC stub creation process is not sound. That is, we relied on the test input. Thus, if a function is not invoked during testing, or the given parameter does not trigger the function to modify the code cache, then we miss this function. Second, to reduce performance overhead and the attack surface, we want to create stubs only for functions that 1) are post-dominated by operations that modifies the code cache; and 2) dominate as many modification operations as possible. Currently, this is done empirically. Though program analysis, we could systematically and more precisely identify these "key" functions. Finally, for the ease of development, our prototype implementation uses shared memory to avoid deep copy of objects when performing RPC. While this strategy is convenient, it may introduce additional cache coherency overhead. With the help of program analysis, we could replace this strategy with object serialization, but only for data that is accessed during RPC.

C. Performance Tuning

In our current prototype implementations, the SDTs were not aware of our modification to their architectures. Since their optimization strategy may not be ideal for SDCG, it is possible to further reduce the overhead by making the SDT be aware of our modification. For example, one major source of SDCG's runtime overhead is RPC invocation, and the overhead can be reduced if we reduce the frequency of code cache modification. This can be accomplished in several ways. For instance, we can increase the threshold to trigger code optimization, use more aggressive speculative translation, and separate the garbage collection, etc.

Second, in our implementations, we used the domain socked-based IPC channel from seccomp-sandbox. This means, for each RPC invocation, we need to enter the kernel twice; and both the request/return data need to be copied to/from the kernel. While this approach is more secure (in the sense that a sent request cannot be maliciously modified), if the request is always untrusted, then using a faster communication channel (e.g., ring buffer) could further reduce the overhead.

Third, we also used the same service model as seccomp-sandbox in our prototypes. That is, RPC requests are served by a single thread in the SDT process. This strategy is sufficient for SDTs where different threads share the same code cache (e.g., Strata), because modifications need to be serialized anyway to prevent data race. However, this service model can become a bottleneck when the SDT uses different code cache for different thread (e.g., JS engines). For such SDTs, we need to create dedicated service threads in the SDT process to serve different threads in the untrusted process.

In addition, our current prototype implementations of SDCG are also not hardware-aware. Different processor can have different shared cache architecture and cache management capability, which in turn affects cache synchronization between different threads. Specifically, on a multi-processor system, two cores may or may not share the same cache. As we have demonstrated, if the translator thread and the execution thread are scheduled to two cores with different cache, then the performance is much worse than when they are scheduled to cores with the same cache. To further reduce the overhead, we can assign the processor affinity according to the hardware features.

VIII. Conclusion

In this paper, we highlighted that code cache injection attack is a viable exploit technique that can bypass many of the state-of-art defense mechanisms. To defeat this threat, we proposed SDCG, a new architecture that enforces mandatory W⊕X policy. To demonstrate the feasibility and benefit of SDCG, we ported two software dynamic translators, Google V8 and Strata, to this new architecture. Our development experience showed that SDCG is easy to adopt and our performance evaluation showed the performance overhead is small.

References

[1] 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.

[2] PaX-Team, "PaX Address Space Layout Randomization," http://pax.grsecurity.net/docs/aslr.txt, 2003. [Online]. Available: http://pax.grsecurity.net/docs/aslr.txt

[3] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh, "On the effectiveness of address-space randomization," in *Proceedings of the 11th ACM conference on Computer and communications security*. ACM, 2004, pp. 298–307.

[4] 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.

[5] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity," in *Proceedings of the 12th ACM conference on Computer and communications security*. ACM, 2005, pp. 340–353.

[6] M. Zhang and R. Sekar, "Control flow integrity for cots binaries," in *USENIX Security Symposium*, 2013.

[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] J. Aycock, "A brief history of just-in-time," *ACM Computing Surveys (CSUR)*, vol. 35, no. 2, pp. 97–113, 2003.

[9] K. Scott and J. Davidson, "Strata: A software dynamic translation infrastructure," *Proceedings of the IEEE*, 2001.

[10] M. Probst, "Dynamic binary translation," in *UKUUG Linux Developers Conference*, vol. 2002, 2002.

[11] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," *ACM Sigplan Notices*, vol. 40, no. 6, pp. 190–200, 2005.

[12] K. Scott, N. Kumar, S. Velusamy, B. Childers, J. W. Davidson, and M. L. Soffa, "Retargetable and reconfigurable software dynamic translation," in *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, ser. CGO '03. Washington, DC, USA: IEEE Computer Society, 2003, pp. 36–47.

[13] F. Bellard, "Qemu, a fast and portable dynamic translator." in *USENIX Annual Technical Conference, FREENIX Track*, 2005, pp. 41–46.

[14] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," in *ACM Sigplan Notices*, vol. 42, no. 6. ACM, 2007, pp. 89–100.

[15] F. Qin, C. Wang, Z. Li, H.-s. Kim, Y. Zhou, and Y. Wu, "Lift: A low-overhead practical information flow tracking system for detecting security attacks," in *Microarchitecture, 2006. MICRO-39. 39th Annual IEEE/ACM International Symposium on*. IEEE, 2006, pp. 135–148.

[16] E. G. Barrantes, D. H. Ackley, T. S. Palmer, D. Stefanovic, and D. D. Zovi, "Randomized instruction set emulation to disrupt binary code injection attacks," in *Proceedings of the 10th ACM conference on Computer and communications security*. ACM, 2003, pp. 281–289.

[17] J. Newsome and D. Song, "Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software," 2005.

[18] W. Cheng, Q. Zhao, B. Yu, and S. Hiroshige, "Tainttrace: Efficient flow tracing with dynamic binary rewriting," in *Computers and Communications, 2006. ISCC'06. Proceedings. 11th IEEE Symposium on*. IEEE, 2006, pp. 749–754.

[19] W. Hu, J. Hiser, D. Williams, A. Filipi, J. W. Davidson, D. Evans, J. C. Knight, A. Nguyen-Tuong, and J. Rowanhill, "Secure and practical defense against code-injection attacks using software dynamic translation," in *Proceedings of the 2nd international conference on Virtual execution environments*. ACM, 2006, pp. 2–12.

[20] J. Hiser, A. Nguyen-Tuong, M. Co, M. Hall, and J. W. Davidson, "Ilr: Where'd my gadgets go?" in *Security and Privacy (SP), 2012 IEEE Symposium on*. IEEE, 2012, pp. 571–585.

[21] P. Pie, "Mobile Pwn2Own Autumn 2013 - Chrome on Android - Exploit Writeup," 2013.

[22] R. H. Netzer and B. P. Miller, "What are race conditions?: Some issues and formalizations," *ACM Letters on Programming Languages and Systems (LOPLAS)*, vol. 1, no. 1, pp. 74–88, 1992.

[23] J. Viega, J.-T. Bloch, Y. Kohno, and G. McGraw, "Its4: A static vulnerability scanner for c and c++ code," in *Computer Security Applications, 2000. ACSAC'00. 16th Annual Conference*. IEEE, 2000, pp. 257–267.

[24] W3C, http://www.w3.org/TR/workers/, 2012.

[25] Google, "Design of chrome v8," https://developers.google.com/v8/design, 2008.

[26] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham, "Efficient software-based fault isolation," in *ACM SIGOPS Operating Systems Review*, vol. 27, no. 5. ACM, 1994, pp. 203–216.

[27] B. Ford and R. Cox, "Vx32: Lightweight user-level sandboxing on the x86." in *USENIX Annual Technical Conference*. Boston, MA, 2008, pp. 293–306.

[28] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar, "Native client: A sandbox for portable, untrusted x86 native code," in *Security and Privacy, 2009 30th IEEE Symposium on*. IEEE, 2009, pp. 79–93.

[29] D. Sehr, R. Muth, C. Biffle, V. Khimenko, E. Pasko, K. Schimpf, B. Yee, and B. Chen, "Adapting software fault isolation to contemporary cpu architectures." in *USENIX Security Symposium*, 2010, pp. 1–12.

[30] M. Castro, M. Costa, J.-P. Martin, M. Peinado, P. Akritidis, A. Donnelly, P. Barham, and R. Black, "Fast byte-granularity software fault isolation," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. ACM, 2009, pp. 45–58.

[31] 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," in *ACM SIGPLAN Notices*, vol. 46, no. 6. ACM, 2011, pp. 355–366.

[32] B. Niu and G. Tan, "Rockjit: Securing just-in-time compilation using modular control-flow integrity," 2014.

[33] CVE, "CVE vulnerabilities found in browsers," http://web.nvd.nist.gov/view/vuln/search-results?query=browser&search_type=all&cves=on.

[34] 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.

[35] 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.

[36] T. M. Austin, S. E. Breach, and G. S. Sohi, *Efficient Detection of All Pointer and Array Access Errors*, ser. PLDI '94. New York, NY, USA: ACM, 1994.

[37] H. PATIL and C. FISCHER, "Low-cost, concurrent checking of pointer and array accesses in c programs," *Software: Practice and Experience*, vol. 27, no. 1, pp. 87–110, 1997.

[38] 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.

[39] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic, "Cets: compiler enforced temporal safety for c," in *ACM Sigplan Notices*, vol. 45, no. 8. ACM, 2010, pp. 31–40.

[40] ——, "Softbound: Highly compatible and complete spatial memory safety for c," in *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '09. New York, NY, USA: ACM, 2009, pp. 245–258.

[41] T. Garfinkel, B. Pfaff, and M. Rosenblum, "Ostia: A delegating architecture for secure system call interposition." in *NDSS*, 2004.

[42] Google, "The sandbox design principles in Chrome." http://dev.chromium.org/developers/design-documents/sandbox.

[43] Microsoft, "App capability declarations (Windows Runtime apps)," http://msdn.microsoft.com/en-us/library/windows/apps/hh464936.aspx, 2012.

[44] Adobe Product Security Incident Response Team, "Inside Adobe Reader Protected Mode," http://blogs.adobe.com/security/2010/11/inside-adobe-reader-protected-mode-part-3-broker-process-policies-and-inter-process-communication.html, 2010.

[45] A. Sintsov, "Writing jit-spray shellcode for fun and profit," 2010.

[46] M. Daniel, J. Honoroff, and C. Miller, "Engineering heap overflow exploits with javascript," in *Proceedings of the 2Nd Conference on USENIX Workshop on Offensive Technologies*, ser. WOOT'08. Berkeley, CA, USA: USENIX Association, 2008, pp. 1:1–1:6.

[47] T. Wei, T. Wang, L. Duan, and J. Luo, "Secure dynamic code generation against spraying," in *Proceedings of the 17th ACM conference on Computer and communications security*. ACM, 2010, pp. 738–740.

[48] C. Kil, J. Jim, C. Bookholt, J. Xu, and P. Ning, "Address space layout permutation (aslp): Towards fine-grained randomization of commodity software," in *Computer Security Applications Conference, 2006. ACSAC'06. 22nd Annual*. IEEE, 2006, pp. 339–348.

[49] R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin, "Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code," in *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM, 2012, pp. 157–168.

[50] V. Pappas, M. Polychronakis, and A. D. Keromytis, "Smashing the gadgets: Hindering return-oriented programming using in-place code randomization," in *Security and Privacy (SP), 2012 IEEE Symposium on*. IEEE, 2012, pp. 601–615.

[51] G. Portokalidis and A. D. Keromytis, "Fast and practical instruction-set randomization for commodity systems," in *Proceedings of the 26th Annual Computer Security Applications Conference*. ACM, 2010, pp. 41–48.

[52] E. G. Barrantes, D. H. Ackley, S. Forrest, and D. Stefanović, "Randomized instruction set emulation," *ACM Transactions on Information and System Security (TISSEC)*, vol. 8, no. 1, pp. 3–40, 2005.

[53] B. Niu and G. Tan, "Modular control-flow integrity," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '14. New York, NY, USA: ACM, 2014, pp. 577–587.

[54] L. Davi, A.-R. Sadeghi, and M. Winandy, "Ropdefender: A detection tool to defend against return-oriented programming attacks," in *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*. ACM, 2011, pp. 40–51.

[55] Y. Cheng, Z. Zhou, M. Yu, X. Ding, and R. H. Deng, "Ropecker: A generic and practical approach for defending against rop attacks." The 21th Annual Network and Distributed System Security Symposium (NDSS'14), 2014.

[56] T. Newsham, "Format string attacks," 2000.

[57] M. Conover, "w00w00 on heap overflows," 1999.

[58] Common Weakness Enumeration, "Cwe-416: use after free."

[59] ——, "Cwe-680: Integer overflow to buffer overflow."

[60] X. Chen, "Aslr bypass apocalypse in recent zero-day exploits," 2013.

[61] F. J. Serna, "The info leak era on software exploitation," *Black Hat USA*, 2012.

[62] G. F. Roglia, L. Martignoni, R. Paleari, and D. Bruschi, "Surgically returning to randomized lib (c)," in *Computer Security Applications Conference, 2009. ACSAC'09. Annual*. IEEE, 2009, pp. 60–69.

[63] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A.-R. Sadeghi, "Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization," in *Security and Privacy (SP), 2013 IEEE Symposium on*. IEEE, 2013, pp. 574–588.

[64] Google, "Seccompsandbox," https://code.google.com/p/seccompsandbox/wiki/overview.

[65] "Yet another new approach to seccomp," http://lwn.net/Articles/475043/, 2012.

[66] "Seccomp and sandboxing," http://lwn.net/Articles/332974/, 2009.

[67] Standard Performance Evaluation Corporation, "SPEC CINT2006 Benchmarks," http://www.spec.org/cpu2006/CINT2006/.

[68] T. V. project authors, http://v8.googlecode.com/svn/data/benchmarks/v7/run.html.