# Dachshund: Digging for and Securing Against (Non-)Blinded Constants in JIT Code

Giorgi Maisuradze
CISPA, Saarland University
Saarland Informatics Campus
giorgi.maisuradze@cispa.saarland

Michael Backes
CISPA, Saarland University
Saarland Informatics Campus
backes@cs.uni-saarland.de

Christian Rossow
CISPA, Saarland University
Saarland Informatics Campus
rossow@cispa.saarland

***Abstract*—Modern browsers such as Chrome and Edge deploy *constant blinding* to remove attacker-controlled constants from the JIT-compiled code. Without such a defense, attackers can encode arbitrary shellcode in constants that get compiled to executable code. In this paper, we review the security and completeness of current constant blinding implementations. We develop DACHSHUND, a fuzzing-driven framework to find user-specified constants in JIT-compiled code. DACHSHUND reveals several cases in which JIT compilers of modern browsers fail to blind constants, ranging from constants passed as function parameters to blinded constants that second-stage code optimizers revert to a non-protected form. To tackle this problem, we then propose a JavaScript rewriting mechanism that removes all constants from JavaScript code. We prototype this cross-browser methodology as part of a Web proxy and show that it can successfully remove all constants from JavaScript code.**

## I. INTRODUCTION

Web browsers continue to be one of the main targets for software exploitation, as demonstrated by the rise of browser-targeting exploit kits [26] and the sheer number of software vulnerabilities discovered in browsers. It is not just the popularity and complexity of browsers that make them an attractive target. Modern browsers also support various scripting languages such as JavaScript and ActionScript. On the one hand, scripting environments have become indispensable to dynamically generate highly-interactive content on the modern Web. On the other, scripting support also allows adversaries to perform prolific attacks. Most notably, in Just-in-Time Return-Oriented Programming (JIT-ROP), an attacker uses the scripting environment to dynamically search for gadgets in existing code (e.g., of the browser or imported libraries) [49]. A viable defense against JIT-ROP attacks is to compile programs in a way that they do not have usable gadgets, e.g., using gadget-free compilation [41] or Control Flow Integrity [55], [56], [14], [37], [55].

However, such protections are typically limited to static code. Consequently, these defenses are ineffective against code spraying attacks [6], in which an adversary leverages scripting environments to dynamically *generate* gadgets (instead of searching for them, like in JIT-ROP). For example, an attacker can embed short gadgets in integer constants of JavaScript code, which the JIT compiler translates to executable shellcode. To protect against dynamically-injected attack code, JIT engine developers and researchers started to rely on *constant blinding*. The goal of constant blinding is to generate code that does not contain user-specified constants. Technically, the JIT compilation process does not emit any constant that may be part of JavaScript statements (such as variable assignments like `a=0x9090`). For example, a simple implementation could remove the constants by XORing two non-predictable values whose XOR result equals to the constant. This way, an adversary can no longer embed shellcode in predictable constants in the JIT-generated code. Consequently, constant blinding has become an important foundation to protect against JIT spraying attacks and is the basis for many other defenses [52], [29]. Most modern browsers such as Chrome or Microsoft Edge (and its predecessor Internet Explorer) deploy constant blinding.

In this paper, we analyze the completeness of constant blinding implementations in JIT engines of modern browsers. We find that a correct and complete constant blinding implementation is not as trivial as it may sound. In fact, browsers typically strive for high efficiency and have to intertwine security defenses with multi-layer optimization schemes. Furthermore, there are dozens of ways to embed constants in JavaScript code, including global and local variable, function parameters, array indexes, bit operations, return statements and many more. As we will show, it is easy to miss some cases, and it becomes a non-trivial challenge to understand the security-related effects of the various optimization layers in JIT engines.

In this context, we propose DACHSHUND, a fuzzing-driven framework that tests the completeness of constant blinding implementations in browsers (or other software with JIT engines, such as PDF readers). The core idea of DACHSHUND is to feed a JIT compiler with JavaScript code snippets that include constants and to trigger the JIT compilation phase(s). We leverage a JavaScript code generator to dynamically generate a high number of diverse code snippets that contain "magic" constants. After JIT compilation, we search for these magic values in the JIT-compiled code in order to test whether the constants have survived blinding. A prototype implementation of DACHSHUND for Chrome and Edge revealed many cases in which the JIT engines of these modern browsers fail to blind user-specific constants, undermining the security guarantees of these implementations.


NDSS '17, 26 February - 1 March 2017, San Diego, CA, USA

http://dx.doi.org/10.14722/ndss.2017.23224

Athanasakis *et al.* have already demonstrated that single-byte or two-byte constants survive the blinding process, as the constant blinding implementations simply do not blind small constants for efficiency reasons [2]. However, we show that the problem of incomplete constant blinding implementations is far more fundamental than JIT compilers skipping over smaller constants. Even blinding all (including smaller) constants would not help to remedy this situation. In fact, all of the surviving constants that we discovered were 32 bits long, giving an attacker full flexibility to embed four-byte gadgets (e.g., any system call).

There are multiple ways to overcome these problems. One approach would be to change the JIT engines of browsers to remedy the situation. However, as we have demonstrated, reaching a complete implementation of constant blinding has proven to be rather difficult and requires modification to each JIT engine separately. Alternatively, we propose to leverage a Web proxy in order to rewrite the JavaScript code before it is delivered to the browser. This way, we can protect any browser behind the proxy without software modifications. Our core idea is to rewrite constants such that they do not appear in the JIT-generated code, regardless of the JIT engine and optimization layer. To this end, we parse the abstract syntax tree (AST) of HTML and JavaScript code, locate any JavaScript constants, and replace them with semantically-equivalent representations that are either not predictable by an attacker, or ideally are moved out of the executable code sections. In addition, we hook critical JavaScript functions (e.g., `eval()`) to remove constants from dynamically-generated JavaScript code. While this approach is clearly less efficient than browser-specific implementations, the average overhead of 22% in JavaScript performance benchmarks is barely noticeable in practice. In addition, rewriting complex JavaScript libraries like jQuery is relatively fast and takes a one-time effort of less than 60 ms. The rewriting outcome can be cached by the client and proxy to eliminate any rewriting overhead in the future, leading to a viable defense scheme in practice.

With this paper, we provide the following contributions:

- We design DACHSHUND, a fuzzing-based framework to search for constants that survive the constant blinding process of JIT engines. DACHSHUND combines code fuzzing techniques with memory carving to discover potentially dangerous blinding leftovers.
- We provide a thorough overview of security deficiencies of the constant blinding implementations in Chrome and Edge, demonstrating that constant blinding by the JIT engines in these browsers is inherently insecure.
- We propose a proxy-based JavaScript rewriting engine that complements existing constant blinding implementations by removing constants from the JavaScript code at an average overhead of 22%.

## II. BACKGROUND

We first provide an overview of the history of code-reuse attacks. We start with Return Oriented Programming (ROP), which clearly demonstrates the general principle behind code-reuse attacks. Next we show a special variant of ROP, called JIT-ROP, which discovers gadgets on-the-fly and evades existing randomization schemes such as ASLR. Besides the attacks, we also discuss potential defenses.

### A. Return Oriented Programming (ROP)

Although ROP was not the first code-reuse attack, it got popular after the wide deployment of Data Execution Prevention (DEP). DEP is a defense technique against a generic stack overflow vulnerability where an adversary writes and executes her shellcode directly on the stack. DEP tackles this problem by marking executable pages non-writable.

As a response to DEP, code-reuse attacks reuse existing code portions of the program instead of injecting new ones. For example, in ret-to-lib(c) [38] an adversary mounts an attack by reusing functions from imported libraries such as libc. This attack was generalized by Shacham [47] with ROP, who proposed to use so-called gadgets (i.e., small sequences of instructions ending with a return instruction) and chain them to get arbitrary program execution. Return instructions are used to chain multiple gadgets together by writing their addresses on the stack. Later, Checkoway *et al.* [9] showed that one can also use any control-flow-changing instruction (e.g., `jmp`, `call`) to achieve the same result.

### B. ASLR vs. JIT-ROP

Address Space Layout Randomization (ASLR) [50] is a widely deployed defense technique against code-reuse attacks. ASLR randomizes the base addresses of the program's memory segments, thus preventing the attacker from predicting the addresses of the gadgets. A remaining weaknesses of this coarse-grained ASLR scheme is that it only randomizes the base addresses of memory segments. Researchers thus proposed fine-grained ASLR schemes that add randomness inside the segment as well [33], [51], [27], [43]. For more details, we refer the reader to Larsen's survey [34].

However, Snow *et al.* proposed a JIT-ROP to overcome ASLR [49]. JIT-ROP is a just-in-time code reuse scheme that follows the assumption that an attacker can repeatedly read arbitrary memory addresses, e.g., via a memory disclosure vulnerability in a scripting environment such as JavaScript. The attacker uses this vulnerability to follow code pointers and collects as many code pages as possible. Next, the attacker searches for desired gadgets (such as Load, Store, Jump) and API function calls (such as LoadLibrary, GetProcAddress) in these code pages. This allows carrying out a just-in-time search for suitable ROP gadgets and thus defeats fine-grained code randomization schemes.

### C. JIT Spraying

While JIT-ROP's idea was to search for *existing* code, it is not guaranteed that the required gadgets actually exist. In fact, Control Flow Integrity schemes may render any gadgets unusable [55], [56], [14], [37], [55], or programs might have been generated by compilers creating gadget-free code [41]. In such a setting, JIT spraying can be used to *inject* attacker-controlled code. JIT-compiled languages, such as ActionScript (Flash) or JavaScript, have become popular in everyday programs such as browsers. Being able to control the input to the compiler (i.e., JavaScript code), an attacker indirectly controls the compilation output. *JIT spraying*, proposed by Blazakis [6], uses this property to evade DEP or ASLR. By repeatedly injecting large amounts of code via attacker-controlled JavaScript objects, the attacker allocates ("sprays")

many executable pages with shellcode. After spraying, the attacker then jumps to an address and hopes that she hits any of the sprayed code pages.

An advanced form of JIT spraying, shown by Athanasakis *et al.* [2], combines JIT spraying and JIT-ROP. Similar to JIT spraying, the authors suggest to craft special JavaScript statements that compile into an attacker-controlled sequence of instructions. For example, JavaScript variable assignments with immediate values (e.g., `var v=0x90909090`) will be compiled into a sequence of assembly instructions containing the instruction that encodes the attacker-supplied immediate (e.g., `mov eax,0x90909090`). Assuming an arbitrary memory read vulnerability, an adversary does not even have to spray many code pages, nor search for existing code (like in JIT-ROP). Instead, she can emit arbitrary gadgets just by controlling constants in JavaScript code.

### D. Constant Blinding

To counter JIT spraying, most browsers have deployed *constant blinding*. This defense technique changes the emitted immediate value by XORing it with a randomly-generated key. For example, instead of compiling the aforementioned JavaScript code into `mov eax,0x90909090`, constant blinding will convert it to the following sequence of instructions:

```
mov eax, (RAND_KEY ⊕ 0x90909090)
xor eax,  RAND_KEY
```

The constant `RAND_KEY` is a randomly generated key, and `(RAND_KEY ⊕ 0x90909090)` is a single integer generated at compile time. Constant blinding thus protects all immediate values with constant-specific keys, and makes the process of JIT spraying highly non-predictable.

A perfect implementation of constant blinding would blind all immediate values in JavaScript code with per-constant keys. In practice (e.g., in MSIE and Chrome), due to performance reasons, only constants larger than two bytes ($> 2^{16}$) are blinded. Although such blinding might seem effective against code spraying, Athanasakis et al. demonstrate that two-byte gadgets are sufficient to mount an attack if they are followed by aligned return instructions (i.e., in the epilogue of the function containing the gadget).

## III. Assumptions

Having discussed the foundation of existing attacks and defenses, we now introduce the threat model and our assumptions on defense techniques that will be considered throughout the paper. These assumptions are in accordance with the environment of other proposed attack techniques [2], [49].

### A. Defense Techniques

We first list the defense techniques that we assume to be deployed in the operating system or the target application:

**Non Writable Code:** We assume that Data Execution Prevention (DEP) is in place, ensuring that the code pages are not writable and thus defending against direct shellcode injections.

**Code Randomization:** We assume that ASLR is enabled in the host operating system, which randomizes the base addresses of the executable and other memory segments every time they are loaded into the memory. Additionally, we assume that fine-grained ASLR is applied to already randomized (by ASLR) memory pages, further complicating the process to guess the address of a gadget.

**Gadget-Free Code:** We assume that static code (i.e., code that is not JIT-compiled) does not contain usable gadgets. For example, this would be the case for gadget-free compilation [41]. Note that JIT-ROP attacks are not possible in such a setting, given the lack of gadgets.

**JIT Defenses:** We assume any defense techniques against JIT spraying that is already present in modern browsers, such as constant blinding (Chrome, Edge) or NOP insertion (Edge). As the main goal of our technique is to emit arbitrary gadgets in the executable code, we assume that sandboxing in the browsers can either be bypassed (e.g., via a vulnerability) or is disabled. For the same reason, we do not consider CFI defenses to be applied to JIT-compiled code.

### B. Threat Model

With these defenses in mind, we now introduce the attacker model. Note that the assumptions listed below are in accordance with existing attack techniques [2], [49].

**Arbitrary Memory Read:** We assume that an adversary is able to read arbitrary *readable* memory of the program. This could be done, for example, by repeatedly exploiting a memory disclosure vulnerability.

**Hijacking Control Flow:** We assume that the target program has a control flow vulnerability that the attacker can exploit to divert the control flow to an arbitrary memory location.

**JIT Compilation:** We assume that the target program incorporates a scripting environment. More specifically, we require that the program has a JavaScript JIT compiler that accepts arbitrary (valid) JavaScript code as input and compiles it to native code. This requirement is met by all modern Web browsers. In principle, our attack is not limited to browsers, as JavaScript is also actively used in other applications (e.g., PDF readers).

## IV. Dachshund: Finding Constants

We now take a closer look at the completeness of the defense technique implementations in JIT compilers of modern browsers. More specifically, we will search for ways, in which the attacker can emit arbitrary gadgets into the executable pages of the browser's memory. To this end, we present DACHSHUND, a fuzzing-based framework that reveals attacker-controllable constants in JIT-compiled code. The basic design of DACHSHUND is shown in Figure 1. The framework consists of a fuzzing component (Section IV-A) that creates diverse JavaScript code snippets to feed them to a JIT compiler for further processing. After JIT compilation, the JIT inspector (Section IV-B) then searches for constants induced by the fuzzer in the executable code pages. The interaction between these two components is steered by the DACHSHUND controller (Section IV-C). In the following, we describe this interplay in more detail.

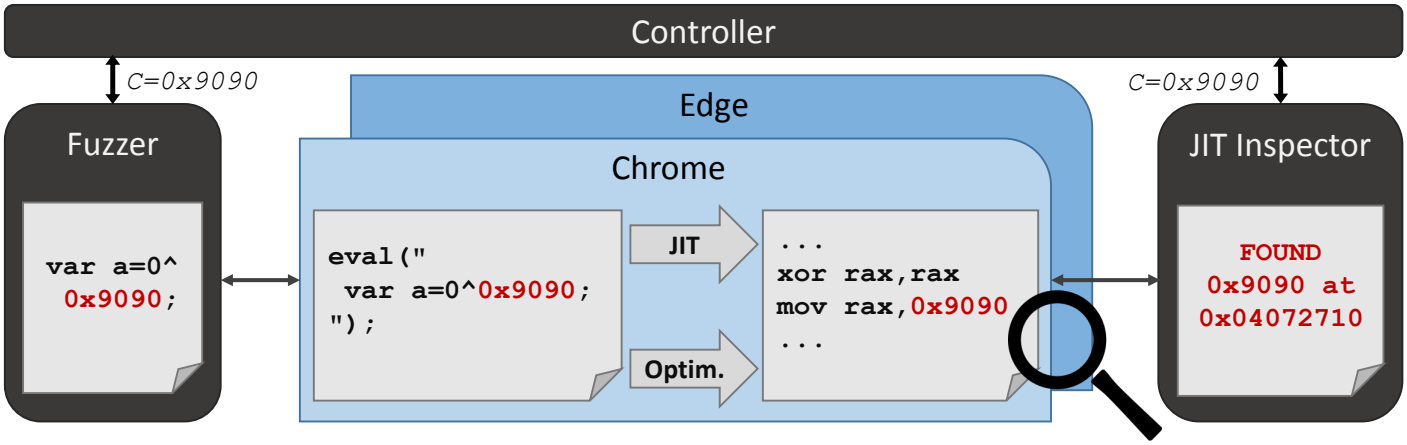


Fig. 1. Overview of the Dachshund architecture and its three components.

### A. *Fuzzing Component*

In the first component of DACHSHUND, we aim to trigger attacker-controllable constants in JIT compiled code. We follow a similar goal to Athanasakis *et al.* [2] and leverage immediate values in JavaScript statements to emit gadgets in the JIT-compiled code. In their paper, the authors exploit the fact that browsers only blind large constants (e.g., Chrome and IE blind values larger than two bytes). We do not limit ourselves to two-byte gadgets and instead challenge the completeness of the constant blinding implementation. That is, we aim to find edge cases in which constant blinding is not applied, or cases where this blinding is reverted by various browser components (such as optimizers).

To search for these edge cases, we leverage code fuzzing. Code fuzzing has a long history as a dynamic testing approach to identify software vulnerabilities [45], [28] (including in browsers). Instead of searching for bugs, we leverage code fuzzing to generate a large diversity of JavaScript code snippets to trigger cases in which constants might not be blinded. Our main idea is to encode "magic" constants in the fuzzed JavaScript code that DACHSHUND's JIT inspector (cf. Section IV-B) can identify.

We implemented our fuzzer based on *jsfunfuzz* [45], a JavaScript fuzzer that is heavily used in testing the Firefox's JavaScript engine. Technically, *jsfunfuzz* generates random JavaScript function bodies (including invalid ones) to test JavaScript engines for vulnerabilities, also covering newly introduced features such as in ECMAScript 6. We extended *jsfunfuzz* to adjust it to our needs: (i) we modified the code generator to reduce the likelihood that code generates syntax errors, and (ii) we increased the chance of large integer immediate values appearing in the generated code. The reason for modification (ii) is straightforward, as we want to test if allegedly-blinded immediate values (i.e., larger ones in the range $[2^{17}, 2^{32})$) are emitted by the compiler. Thus, we want to maximize their incidence in the generated JavaScript code. Modification (i) is required to reach the compilation stage, which will not be the case if the generated JavaScript code contains a syntax error. This again highlights the difference between our motivation for code fuzzing and the typical motivation for triggering software vulnerabilities.

We feed the JavaScript code snippets that are generated by the fuzzing component to two popular browsers: Microsoft Edge and Google Chrome (and their corresponding JavaScript engines: Chakra and V8, respectively). We exclude Mozilla Firefox from our experiments, as its JavaScript engine does not implement constant blinding.

### B. *JIT Inspector Component*

The JIT inspector component relates integer constants in randomly generated JavaScript code to the sequence of bytes representing the same number in the JIT-compiled machine code. Technically, we attach to the renderer process of the browser and inspect its code pages created at runtime. Once the magic value encoded by the fuzzing component is found, the JIT inspector has likely found a constant that has survived the blinding phase.

However, to fully understand *when* to inspect the code pages, it is important to note that JavaScript engines implement multiple levels of compilation. Typically, the first-level JIT compiler is fast but produces low-performance code, which is then optimized by a second-level JIT compiler if it has been executed frequently. We refer to the first level compiler as a baseline compiler and the second level as an optimizing compiler. In our experiments, we consider the code generated by both compilers, as the attacker has full control of triggering either of the two compilers by carefully choosing how often she executes a piece of code.

A distinction between Chrome and Edge has to be made when the compilers kick in. Edge has an interpreter that interprets the JavaScript code until it becomes warm (i.e., when it is executed around 50 times). Only after that, a JavaScript function is compiled by the baseline compiler. In contrast, Chrome skips the interpreting step and directly compiles the JavaScript function upon first execution. Consequently, to trigger a baseline compilation of a JavaScript function, one has to call the function once for Chrome and 50 times for Edge—again, a parameter that is under full control of the attacker. In both browsers, a baseline-compiled JavaScript function is recompiled by the optimizing compiler after it becomes hot (i.e., after it is executed over 1000 times). The optimizing compiler leverages code analysis techniques to produce highly

efficient code (e.g., by incorporating inferred type information or function inlining). To trigger an optimization of a JavaScript function, one has to call it more than 1000 times. However, given the runtime of short JavaScript functions, this is not a practical burden to attackers, i.e., it can be optimized in a matter of milliseconds.

Putting all this together, the basic algorithm of the JIT inspector is the following:

**(J1)** The JIT inspector receives a set of integers (the magic values) as an input that has to be found in the JIT-compiled code.
**(J2)** It attaches itself to the required renderer process of the tested browser (i.e., the correct browser tab containing the tested JavaScript code).
**(J3)** By looking at the permissions of the memory pages, the JIT inspector retrieves a set of pages that were generated by the JIT compiler. It does so by scanning for pages with RWX protection in Chrome and RX protection in Edge.
**(J4)** Functions in these code pages are separated by `0x00` bytes in Chrome and `0xcc` (`int3`) bytes in Edge. Therefore, starting from a page boundary, the JIT inspector can easily identify all functions, and extracts the corresponding machine code.
**(J5)** As a final step, the JIT inspector searches for the input integers (J1) in the machine code. In case of a match, the JIT inspector returns the disassembly of the function that contains the constant(s).

Note that in the last step (J5), where we search for the integer values in the machine code, we may encounter false positives. That is, machine code may accidentally contain the value that we searched for, which was however not a consequence of the JavaScript code. We can deal with false positives in two ways: (i) We can manually inspect the disassembled output of the machine code to verify that it indeed corresponds to a JavaScript statement, or (ii) we can reuse the same JavaScript function with a different set of immediate values, and check if we get the match again. For the sake of simplicity, we used the first approach and manually inspected all constants found by DACHSHUND, while the latter solution is a fully-automated way to exclude any chance of false positives.

### C. Controller Component

As a third and last component, we add a controller that steers the interplay between the fuzzer and inspector components. The goal of the controller is to steer synchronization between fuzzer and inspector. The controller does so in the following repeating steps:

**(CC1)** The controller instruments the fuzzing component to generate a textual representation of a new JavaScript function (`jsfunStr`).

**(CC2)** Using `eval`, the controller generates a JavaScript function from `jsfunStr` (`jsfun=eval(jsfunStr)`).

**(CC3)** If `eval` fails (i.e., `jsfunStr` has a syntax error), return to step (CC1). Otherwise, the controller compiles `jsfun` by calling it either once (Chrome) or fifty times (Edge), triggering the respective baseline compilers.

**(CC4)** The controller then triggers the JIT inspector to find constants that survived blinding. It passes all constants that are in the JavaScript code generated in (CC1) to the JIT inspector. If the JIT inspector returns positive matches, these are logged accordingly.

**(CC5)** The controller then triggers the optimization compiler on `jsfun` by calling the function 2000 times and repeats step (CC4) on the optimized code.

### D. Experimental Results

After implementing DACHSHUND for Edge and Chrome, we experimented to test the constant blinding efficacy of these two browsers. We ran DACHSHUND in a VirtualBox virtual machine, running Windows 10 on an Intel i5-4690 CPU having 3.50 GHz and 8 GB RAM. We ran each experiment for two hours per browser. In this time, DACHSHUND detected 124 constants in Chrome and 58 in Edge. Some of these results contained similar JavaScript statements involving emitted constants; therefore, we manually filtered them to get unique cases only, which resulted in 22 different cases in Chrome and 21 in Edge. We manually verified these cases and in all instances found a true positive, i.e., we successfully found a non-blinded constant. In Chrome, constants were only emitted by the optimizing compiler, while in Edge constants were found in both baseline and optimizing stages. The summarized outcome of the experiments is that many JavaScript constants are directly emitted into machine code—despite constant blinding. In the following, we will categorize these cases into classes of constants that bypassed the blinding process.

Experiment results from both of the browsers showed that a major source of constants were arguments to `Math` functions. `Math` is a built-in JavaScript object, containing basic mathematical functions and constants. Immediate values passed as an argument to `Math` functions (like `Math.round(0x1234)`) end up in the JIT-compiled code. Manual verification showed that the optimizing compiler of Chrome also emits constants when calling any other functions, such as built-in functions of JavaScript (e.g., `Array.push(...)`) or even user-defined ones. In assembly, these constants are emitted when argument registers are set or when arguments are pushed on the stack. Consequently, calling a function with more parameters (e.g., `Math.max(X, Y)`) or calling them multiple times emits more constants.

In Edge, however, the situation is different. Manual verification showed that all the emitted constants (not only from function calling) are coming from the same assembly instruction, namely storing the constant into a register. Moreover, this instruction is always located at the beginning of the function, after the prologue, and not where the actual statement (involving the constant) is compiled. This is likely caused by a caching mechanism of Edge, which stores an immediate value in an unused register to use it later in a function.

For example, consider the following JavaScript code:

```
function jsfun() {
  return Math.trunc(0x12345678);
}
```

Chakra, Edge's JIT engine, will compile this statement into the following sequence of assembly instructions:

```
...          ; prologue
mov  rsi,0x10000[12345678] ; Emitted constant
...          ; Other function code
mov  r9, rsi ; Setting Math.trunc parameter
...          ; Setting other parameters
call r12     ; Call Math.trunc
...          ; Other code + epilogue
```

As it can be seen, the constant `0x12345678` is emitted as part of a 64-bit constant. Note that Edge uses 48-bit values for constants. Thus, the least significant bit of the first two bytes denotes the *tag bit* and indicates type of the encoded value, that is an integer constant in our example. The instruction `mov rsi,XXX` is the integer constant caching behavior of Edge, which we mentioned earlier. Interestingly, Edge uses the cached integer value not only when the constant value itself is used, but also when other (similar) integers are used. For example, to set an integer constant `0x12340000` in an `rax` register, Edge utilizes the cached value and emits the following code:

```
lea rax,[rsi-0x5678] ;set rax to 0x12340000
```

The difference between the cached and target value is encoded in `lea`. If needed, this can be further exercised by an attacker to emit more than one constant per function.

Summing up, Edge emits constants in both phases of compilation (baseline and optimizing), but emits only one constant per function, located at the beginning of the compiled function. This does not limit the attacker, as she is able to compile many small functions to emit multiple gadgets. In contrast, Chrome's JavaScript optimizer emits integer constants as part of the compiled statement involving the constant, and thus can be used multiple times to emit many constants in the same function

In general, DACHSHUND found many more ways to embed integers that survive blinding. Other non-blinded JavaScript statements include: ternary operators (`c?0x12345678:0x9abcdef`), return statements (`return 0x12345678`), cases of a switch statement (`case 0x12345678:`), a bit-wise operations (`i=j^0x12345678`), writing an integer to a global variable (`glob=0x12345678`), or to an array element (`arr[0]=0x12345678`). Figure 2 shows the aforementioned gadget emitting statements in Chrome and their corresponding x86 code after compilation. This demonstrates that popular constant blinding implementations are far from complete, as many typical code constructs are not touched by the compiler—not even the textbook JIT spraying example of variable assignments.

### E. Proof-of-Concept Gadget Generation

As a final step of our evaluation, we leverage the previously-observed shortcomings in constant blinding implementations in order to create JavaScript functions that emit meaningful gadgets into the executable memory.

| | |
|---|---|
| `m = i ? 0x12345678 :`<br>`0x23456789` | `0 test rax,rax`<br>`1 je 4`<br>`2 mov ebx,23456789h`<br>`3 jmp 5`<br>`4 mov ebx,12345678h` |
| `switch(j){`<br>`case 0x23232323: m++;`<br>`}` | `0 mov rdx,[rbp+20h]`<br>`1 cmp edx,23232323h`<br>`2 jne XXX` |
| `0x34343434[j]` | `0 mov rdx,3434343400000000h`<br>`1 ;set other parameters`<br>`2 call GetProperty` |
| `m = j ^ 0x45454545` | `0 mov rax,[rbp+20h]`<br>`1 xor eax,45454545h` |
| `globvar = 0x56565656` | `0 mov rax,1AF729D6001h`<br>`1 mov r10,5656565600000000h`<br>`2 mov [rax+0Fh],r10` |
| `globarr[i] = 0x67676767` | `0 mov [rdx+XXX],67676767h` |
| `return 0x12121212` | `0 mov rax,1212121200000000h` |

Fig. 2. Gadget emitting JavaScript statements in Chrome and their corresponding disassembly after rewriting.

For demonstration purposes, we inject the same set of gadgets that was used by Athanasakis *et al.* [2] to set the argument registers for the VirtualProtect function:

```
pop r8,  ret ;  4158 c3
pop r9,  ret ;  4159 c3
pop rcx, ret ;    59 c3
pop rdx, ret ;    5a c3
pop rax, ret ;    58 c3
```

*1) Chrome:* In Chrome, we created the following single JavaScript function containing the immediate constants that correspond to the required gadgets:

```
function chromeGadgets() {
 globar[0] = 0xc35841;
 globar[1] = 0xc35941;
 globar[2] =-0x3ca7a5a7;
}
```

As we have seen, writing an immediate constant to an array element emits it to the JIT code after compilation. Therefore, in `chromeGadgets`, we write the required constants into `globar`, which is a global array declared outside the function. Note that the order of the bytes are swapped in integer constants because of the little-endian format of the underlying x86 machine. Furthermore, to also use the most significant bit in the last gadget, we use a negative number `-0x3ca7a5a7` that will be represented in binary as `0xc3585a59`. After executing this function multiple times, i.e., triggering the optization, the optimizing compiler of Chrome generates the following sequence of instructions:

```
mov [rbx+0x1B], 0x00[C35841] ; c7431b[4158c3]00
mov [rbx+0x23], 0x00[C35941] ; c74323[4159c3]00
mov [rbx+0x2B], 0x[C3585A59] ; c7432b[595a58c3]
```

*2) Edge:* In Edge, given constant caching, we had to create three separate functions to generate the required set of gadgets (note that this is not a limitation as we are not constrained by the maximum number of created functions):

```
function r8(){ Math.trunc(0xc35841); }
function r9(){ Math.trunc(0xc35941); }
function racdx(){
 Math.trunc(-0x3CA7A5A7);
}
```

Triggering the compilation of each of these functions, i.e., calling them 50 times, resulted the required gadgets at the beginning of the corresponding functions. The following is the disassembly of the instructions emitting the gadgets:

```
mov rsi,0x1000000C35841; 48be4158c30000000100
mov rsi,0x1000000C35941; 48be4159c30000000100
mov rsi,0x10000C3585A59; 48be595a58c300000100
```

## V. Defending Against Constants

DACHSHUND has revealed that major browsers are susceptible to emitting attacker-controlled four-byte values into executable code. Even though Chrome and Edge deploy constant blinding to defend against gadget emission, their implementation is still not complete. While it was already known that constant blinding implementations emit two-byte gadgets [2], our automated DACHSHUND framework discovered that even four-byte integer constants are emitted in certain scenarios.

There are several options to solve the aforementioned problems. The naïve and likely the most efficient solution would be to modify the JavaScript engines in the browsers to incorporate constant blinding in all missing cases (e.g., inlining integer constants in Chrome's optimizing compiler or preloading registers in Edge). This would remove the problem of arbitrary four-byte gadget generation, presumably without too much overhead. However, to also get rid of two-byte gadgets, constant blinding schemes in the browsers must be extended to cover integer constants of all sizes, significantly degrading the performance [2]. In addition, changing the JIT compiler is not always possible, especially in closed-source browsers; at the very least, it requires compiler-specific engineering effort to cover all browsers.

Alternatively, we propose to randomize the JavaScript code before the code is delivered to the browser. As DACHSHUND identified, the main source of gadgets in JIT-compiled code is inlined or cached integer constants. Consequently, the main idea of our defense is to remove these constants by rewriting the JavaScript code. We prototype our technique as part of a Web proxy that mediates Web traffic between clients and servers. Once implemented, our solution protects any client behind the Web proxy. One could also implement the same approach as a browser extension to target specific browsers separately. Browser-aware implementation can be optimized to only rewrite the parts of the JavaScript that are attacker-controllable in the specific browser, thus reducing the performance overhead caused by the rewriting. However, as our main goal was to prove the efficacy (and not efficiency) of a solution based on JavaScript rewriting, we opted for a proxy-based rewriting that is agnostic to the specific browsers.

The possible downside of a proxy-based solution is that we rely on all clients in a network to use a Web proxy for browsing. This also means that the proxy has to intermediate HTTPS traffic and thus provides custom certificates for HTTPS communication between the browser and the proxy. While this might sound cumbersome, most corporate proxy vendors offer such capability. HTTPS traffic inspection is *de facto* standard in many organizations that leverage next-generation firewalls, such as Baracuda Networks [4], Forcepoint [21], Palo Alto Networks [42], MS Forefront [22], Blue Coat [7], Fortigate [23], Zscaler [57]. We will discuss this in more detail in Section VI. Note, however, that the design choice of *where* to deploy JavaScript-based rewriting can be changed depending on the needs.

### A. Basic Idea

The core of our idea is to rewrite JavaScript code into semantically equivalent code that does not contain any integer constants. There are several alternatives for how integer constants can be replaced. A simple example of such replacement would be to split an integer constant into parts (similar to constant blinding), changing the constant `X` into `YoZ`, where `o` is any JavaScript operation such that `YoZ=X`. However, as we modify the JavaScript code, this operation would be easily folded by the compiler and `X` would still be emitted. Another solution is to generate a new `Number` object every time a constant is used, e.g., via `parseInt`, which takes a string representation of a number as an input and outputs its corresponding `Number` object. This replacement would transform a constant `X` into a statement: `parseInt('X')`. A drawback of this method is that it executes a `parseInt` function call every time an integer constant is used, thus greatly decreasing performance. In the following, we show how this can be optimized.

In our prototype, we hide integer constants by replacing them with global objects. For example, a JavaScript statement `var i=1234` will be replaced by the following pair of statements:

```
window.__c1234=parseInt('1');
window.__c1234=parseInt('1234');
```

These statements will be prepended at the beginning of the script. During the initialization of these global variables, we use `parseInt` such that the assignment does not emit the constant. In the case of a call to `parseInt`, the argument is a string and therefore only the reference to that string (and not its value) will be emitted to the executable compiled code. Additionally, as it is seen in the example, we initialize the same object twice: first with some random number, and second with the original value. This is necessary to trick the optimizer into thinking that the value of the global object is changing, otherwise the global integer will be inlined into the compiled code. This modification shows the intuition behind our defense: First, by replacing integer constants with global objects, we get rid of integer literals from JavaScript code, which is the main reason of gadget-emission in Edge; And second, we mark these global objects as volatile (i.e., they can be modified by other parties at any point) by setting their values multiple times. This will force the optimizer to resolve their values at runtime instead of inlining them into the code, successfully

removing the sources of gadgets in Chrome. We have manually verified that compilers replace neither `window.__c1234` nor `parseInt('1234')` with the integer 1234 in none of the browsers.

However, removing constants from JavaScript code is a little more complex than that. Because JavaScript has implicit conversion between types, which can also be inlined, e.g., by the optimizing compiler of Chrome. Therefore, we have to additionally protect against possible implicit type conversions. For example, a JavaScript statement `var i='1234'&5678` will also emit 1234 as an integer constant. We handle these cases by finding all strings that can be implicitly converted to integers and call the `toString` method on them (`var i=('1234').toString()&5678`). This returns a new string object every time it is called and therefore is not optimized. Other string methods (such as `substr`) can also be used as an alternative. There are other possibilities in JavaScript of implicit type conversions to integers, e.g., from Boolean to integer (true→1, false→0), from Array to integer ([]→0). To trigger the conversion, these objects must be used as a part of arithmetic operations, which will then try to convert them to the most reasonable integers. We will discuss these cases in Section VI.

To eradicate all integer constants, we rewrite all possible places where JavaScript code can be written. We distinguish between the following five cases:

**(C1)** An external JavaScript file referenced using a src attribute of an HTML script tag, such as:
```
<script src="jsfile.js"></script>
```

**(C2)** JavaScript inside an HTML script tag, such as:
```
<script>/*JS code*/</script>
```

**(C3)** Inline event handlers, defined inside HTML tags, e.g.:
```
<img onclick="/*JS code*/" />
```

**(C4)** Dynamically created JavaScript code, e.g., by using one of the following methods:
```
eval("/*JS code*/")
Function("/*JS code*/")
setTimeout("/*JS code*/", 0)
setInterval("/*JS code*/", 0)
```

**(C5)** Dynamically created HTML nodes, which an attacker might use to inject new JavaScript code, such as:
```
head.appendChild(/*DOM node*/)
el.innerHTML="<script>/*JS code*/</script>"
```

In the following, we will describe the implementation details of how we actually handled these cases.

### B. Implementation Details

We implemented our prototype in *Node.js* [39], using the *http-mitm-proxy* package [30] as a basis for an HTTP proxy. To identify all constants in JavaScript code, we use *Esprima* [19], a JavaScript parser with full support for ECMAScript 6. We leverage the abstract syntax tree (AST) to identify integer constants or string constants representing numbers. We leverage *Estraverse* [20] to traverse the AST and replace AST nodes (e.g., replacing number literals with global objects). Finally, we use *Escodegen* [18] to generate JavaScript code that corresponds to the updated AST.

The general workflow of the rewriter can be summarized in the following steps:

**(RW1)** The rewriter takes JavaScript code as input and derives its AST.

**(RW2)** The rewriter traverses the generated AST. For each literal node (i.e., integer or string immediate values), the rewriter distinguishes the following cases:
- Integer constants (e.g., `123`) are replaced with a node corresponding to the statement (e.g., `window.__c123`). Then, the rewriter adds initialization code for this node (e.g., `init+='window.__c'+ 123+'=parseInt("'+123+'");'`)
- String constants representing numbers (e.g., `'1234'`) are replaced with an AST node of the statement: `('1234').toString()` to avoid implicit casts to (possibly constant) numbers.

**(RW3)** Finally, the rewriter generates JavaScript code that corresponds to the updated AST, notably including the global variables' initialization scripts.

The JavaScript rewriter becomes an integral part of the Web proxy. That is, we modify responses from server to client (i.e., browser). If the response is a JavaScript file (C1), we directly return the rewritten result to the client. In case of an HTML file, we extract and rewrite inline JavaScript between script tags (C2) and inline event handlers (C3). For dynamic code (C4), we inject new JavaScript code as the first element of the head tag, which hooks the dynamic code generator functions (e.g., `eval, Function, setTimeout, setInterval`) and dynamically rewrites the code (i.e., the first argument of these functions) before calling the original function. For dynamic HTML elements (C5), we attach a mutation observer to the document object. This allows us to react to DOM tree modifications by the attacker. For each node that is modified in the DOM tree, we check if it is a script tag or if it contains a script tag in its child nodes, and if so, we extract and rewrite its JavaScript content.

Note that in order to rewrite dynamically generated code (e.g., for (C4) and (C5)), we use synchronous `XMLHttpRequest` requests from our hooked JavaScript functions and mutation observer to the proxy. The JavaScript code that needs to be modified is added to the request. The response from the proxy contains the rewritten JavaScript code.

### C. Evaluation

In the following, we evaluate our implemented defense technique. First and foremost, we test the efficacy of the solution and apply DACHSHUND to reveal if there are remaining attacker-controlled constants in the JIT-emitted code. Second, we evaluate the performance overhead of the proposed solution. As we rewrite the JavaScript code that is executed in a browser, we consider two sources of overhead: (i) the overhead caused by rewriting JavaScript code, and (ii) the performance overhead of the rewritten JavaScript code, running inside a browser. We evaluate the latter in Google Chrome 50 and Microsoft Edge 25. The underlying system is Windows 10 running on an Intel Core i7-2670QM machine with 2.20GHz frequency and 6GB RAM.

| | |
|---|---|
| `m = i ? __c12345678 : __c23456789` | `0 test rax,rax`<br>`1 je 5`<br>`2 mov rbx,&__c12345678`<br>`3 mov ebx,[rbx+13h]`<br>`4 jmp 7`<br>`5 mov rbx,&__c23456789`<br>`6 mov ebx,[rbx+13h]` |
| `switch(j){`<br>`case __c23232323: m++;`<br>`}` | `0 mov rbx,&__c23232323`<br>`1 mov ebx,[rbx+13h]`<br>`2 cmp edx,ebx`<br>`3 jne XXX` |
| `__c34343434[j]` | `0 mov rax,&__c34343434`<br>`1 mov eax,[rax+13h]`<br>`2 ;set other parameters`<br>`3 call GetProperty` |
| `m = j ^ __c45454545` | `0 mov rbx,&__c45454545`<br>`1 mov ebx,[rbx+13h]`<br>`2 mov rdx,[rbp+20h]`<br>`3 xor ebx,edx` |
| `globvar = __c56565656` | `0 mov rax,&__c56565656`<br>`1 mov edx,[rax+13h]`<br>`2 mov rax,&globvar`<br>`3 mov [rax+0Fh],rdx` |
| `globarr[i] = __c67676767` | `0 mov rax,&__c67676767`<br>`1 mov eax,[rax+13h]`<br>`2 mov [rbx+XXX],eax` |
| `return __c12121212` | `0 mov rax,&__c12121212`<br>`1 mov eax,[rax+13h]` |

Fig. 3. Gadget emitting JavaScript statements in Chrome and their corresponding disassembly after rewriting. `&__cXXXXXXXX` denotes the address of the corresponding JavaScript global variable.

*1) Rewriting Efficacy:* First, we evaluate the correctness of the rewriter to see if all integer constants are indeed eradicated from the JIT-compiled code. Therefore, we tested the rewriter against all JavaScript functions found by DACHSHUND. Initially, we verified that all these functions actually emitted integer constants, i.e., we did not get any false positives from DACHSHUND. We found that all 22 different functions in Chrome and 21 in Edge did emit integer constants into the code. We then modified these functions with our JavaScript rewriter and ran the experiment again. After rewriting, none of the JavaScript functions emitted any integer constants in the JIT code, neither for Chrome nor Edge, proving the completeness of the rewriter. Figure 2 shows the disassembly of the native code of the gadget-emitting statements in Chrome, whereas Figure 3 shows the same statements and their disassembly after applying our rewriter.

The code examples, found by DACHSHUND, cover only directly used JavaScript integer constants. While this is sufficient for Edge, where the source for emitted gadgets are integer caching, optimizing compiler of Chrome can still inline the values after implicit conversion. To test the rewriting efficacy of implicit constants (i.e., from string objects to integers in our case), we did manual verification. More specifically, we created JavaScript functions containing string literals that are implicitly converted to integer types. After rewriting, all these string literals were converted to string objects (via invoking toString on them), and thus did not emit any integer values. However, strings are not the only JavaScript objects that are implicitly converted to integers. For example, Hieroglyphy [32] uses conversion between arrays (`[...]`) and objects (`{...}`) to integers. Using these conversions inside the function does not emit attacker-controlled values. However, they can be used by the attacker to initialize a global variable and then use the global variable inside the function to inject the required value. Because the global variable will be initialized once, by the attacker, it will be inlined into the code (by Chrome), emitting the gadgets. This problem can be solved using a similar technique that we used before. That is, we can replace global variable initializations in the code by initializing the global variable with a random number first, and then setting it to the original value. This way, optimizer will have to resolve the value of the global variable at runtime and will not be able to inline it into the code. Although we manually verified that this modification indeed removes the attacker-controlled values from the code, it is not included in the current implementation of our defense scheme.

| | |
|---|---|
| `m+=0x12000000\|`<br>`0x00340000\|`<br>`0x00005600\|`<br>`0x00000078;` | `0 mov ebx, [rbx+13h]`<br>`1 add ebx,12345678h` |
| `m+=__c12000000\|`<br>`__c00340000\|`<br>`__c00005600\|`<br>`__c00000078;` | `0 mov rax,&__c12000000`<br>`1 mov eax,[rax+13h]`<br>`2 mov rbx,&__c00340000`<br>`3 mov ebx,[rbx+13h]`<br>`4 or ebx,eax`<br>`5 mov rax,&__c00005600`<br>`6 mov eax,[rax+13h]`<br>`7 or eax,ebx`<br>`8 mov rbx,&__c00000078`<br>`9 mov ebx,[rbx+13h]`<br>`10 or ebx,eax` |

Fig. 4. Constant splitting in JavaScript (Chrome) before and after rewriting.

Other obfuscation techniques of JavaScript code also contain integer splitting to hide their values. For example, instead of having a single constant `var i=0x12345678`, the attacker might try to split it (e.g., into separate bytes): `var i=0x12000000|0x340000|0x5600|0x78`. After optimization, these constants will be folded by the compiler into a single integer and will be emitted into the JIT-code. However, our rewriter will turn each of these constants into global objects, forbidding both constant folding and inlining (Figure 4).

*2) Rewriting Overhead:* To evaluate the overhead of the JavaScript rewriter, we chose to measure the rewriting overhead of two popular and large JavaScript libraries, jQuery (version 2.2.3) and AngularJS (version 1.5.5). These libraries are commonly embedded in typical Web applications and are relatively large compared to other custom JavaScript implementations (jQuery has 259 kB, AngularJS 1.1 MB). Moreover, both of these libraries also provide the compressed (i.e., "minified") versions to reduce the download size (jQuery 86 kB, AngularJS 158 kB). For the evaluation, we rewrote these libraries (both compressed and uncompressed) 200 times. We measured the time required to rewrite these libraries, including all steps (RW1) up to (RW3). The results of the evaluation are presented in the following.

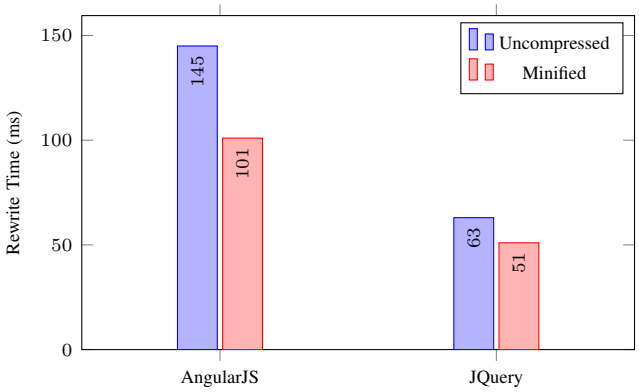


Fig. 5. Averaged times for rewriting JavaScript libraries

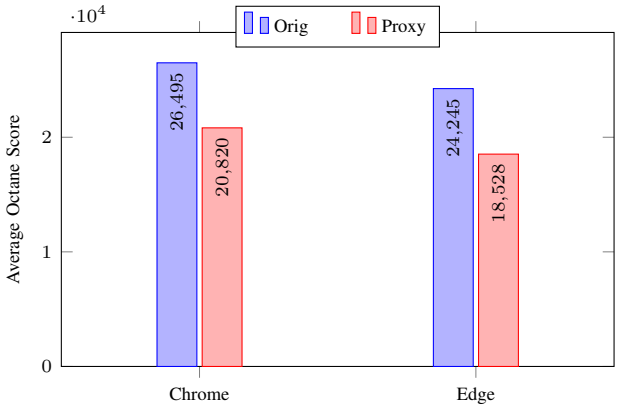


Fig. 6. Averaged Octane scores in Chrome and Edge

The minified versions of the libraries took less time to be rewritten. On average, rewriting AngularJS took 145 ms and 101 ms for the uncompressed and minified versions, respectively. Rewriting jQuery took 63 ms and 51 ms, respectively (see Figure 5). We argue that this overhead of a mere 100 ms is acceptable to typical Web users, as network latencies and bandwidth constraints are more significant when loading these libraries. In addition, note that rewriting is a one-time effort and the rewritten JavaScript library can be cached by the proxy as well as on the client side. Such caching mechanisms are part of COTS browsers and require no further client software modifications. Finally, rewriting multiple scripts simultaneously is an effort that can be trivially parallelized to further improve performance.

*3) Runtime Overhead:* Next, we evaluate the runtime overhead that is incurred on the client side due to the modified JavaScript code. To this end, we leverage Octane, a commonly-used benchmark framework for JavaScript engines [40]. For the evaluation, we took the averaged scores from 5 runs of Octane benchmarks. JavaScript runtime showed the following results: Figure 6 illustrates the performance comparisons between the original and the rewritten engine. The unit is the score measured by the Octane benchmarks, and higher is better. On average, we measure 21% performance decrease in Chrome and 24% in Edge. The average overhead is significant, but performance is mainly degraded by a few outliers in the benchmark suite, such as:

**zlib** In order to test the performance of the compiler, zlib uses `eval`. This causes the *rewrite time* of our rewriter to be added to the runtime of the script, as the code passed to `eval` needs to be rewritten dynamically. Additionally, the compiled zlib script extensively uses integer constants that further degrade the performance.

**CodeLoad** This benchmark measures how quickly a JavaScript engine can execute a script after loading it. CodeLoad uses `eval` to compile JQuery and Closure libraries and therefore again includes the rewriting time.

While the use of dynamic code (like in `eval`) degrades performance, we cannot exclude such code from our rewriter, as it would give a possibility to the attacker to enter constants using dynamic code. However, the experiments have shown that it is mainly dynamic code rewriting that causes performance impacts, and libraries that do not leverage such dynamic code have an acceptable overhead. Without the two poorly-performing benchmarks, the overhead decreases to 12% in Chrome and 13% in Edge. Note that the overhead of popular libraries could be eliminated by whitelisting (and thus not rewriting) trusted scripts, as our threat model is only relevant to non-trusted and attacker-controlled JavaScript inputs. Alternatively, our rewriter could cache popular libraries after they have been already rewritten to provide them to the client without any rewriting overhead.

To put things into perspective, we now compare the performance of our scheme with the performance of a non-optimized JIT compiler. The intuition here is that our suggested attack technique against Chrome relies on abusing output of the optimizing compiler. Disabling the optimizing compiler thus is a viable alternative to protect against attacker-induced gadgets. Therefore, we performed another experiment and also included Chrome with a disabled optimizing compiler (by running Chrome with the V8 flags `noopt` and `nocrankshaft`). Figure 7 shows the complete list of all Octane benchmarks, running in three modifications of Chrome: (i) original, (ii) original with proxy (i.e., rewritten JavaScript), and (iii) with the optimizing compiler disabled. As can be seen, with the exception of the two libraries that require rewriting of dynamic code, our proposed solution outperforms the disabled optimizer by around a factor of eight and thus seems to be the preferable option.

Although the overhead for dynamic scripts seems significant, our JavaScript rewriter usually completes in a matter of milliseconds. Rewriting a JavaScript library as big as jQuery takes, on average, less than 60 ms (see Figure 5). This can be further improved by incorporating caching in our proxy, using hashes of the dynamic script as a key. This way, for example, when compiling a jQuery library 100 times using `eval` (e.g., as done in CodeLoad), the rewriter spends 60 ms on the initial request and serves the subsequent requests without any delay caused by the rewriting process.

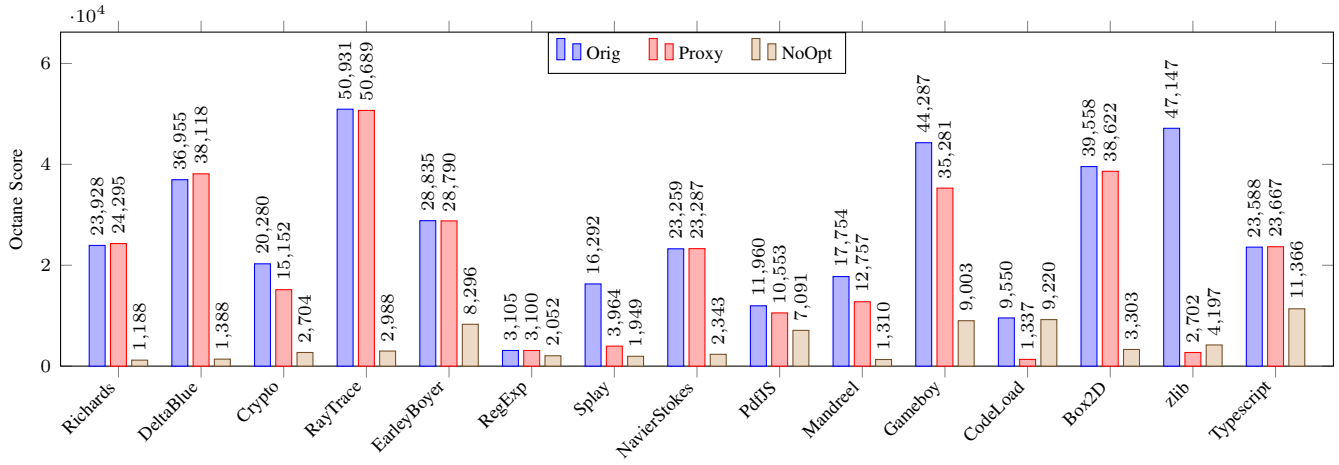


Fig. 7. Octane results in Chrome (Original vs Proxy vs Not-Optimized)

## VI. Discussion

In this section, we revisit the two proposed frameworks, and discuss their implications and possible limitations.

### A. Implications of Dachshund

DACHSHUND has revealed several ways attackers can inject arbitrary long gadgets into JIT-compiled code. We will now discuss how this is relevant from a security perspective.

**How bad is attacker-induced shellcode, really?** One could argue that additional defense schemes in browsers will protect against control-flow diversion attacks, regardless of whether an attacker can inject shellcode or not. While this argument is not wrong *per se*, we believe that our findings have important implications anyway. First, constant blinding is part of the two most popular browsers, and—as we find—a security feature that can be easily circumvented by attackers. Relying on such schemes is only helpful if they are also implemented correctly. Otherwise, as we find, they give security guarantees that do not hold in practice. Second, attacks in the past have demonstrated that even additional security mechanisms such as CFI or sandboxing cannot protect against successful browser exploitation. We thus argue that it is not an "either-or" question which security mechanisms to use; we see the need for complementary techniques to defend against browser threats. Finally, predictable gadgets may also have severe security implications on even stronger threat models. For example, assuming that the location of gadgets can be identified, schemes that propose non-readable code (such as Execute-no-Read proposals [3], [12], [13]) can potentially be evaded. We will perform such an evaluation in future work.

**Does it matter that you found four-byte constants?** DACHSHUND is not the first work to target constant blinding schemes. Athanasakis et al. [2] had already proven that two-byte constants are sufficient to assemble suitable ROP chains. However, we looked at the problem from a different angle. Instead of using constants that are *excluded* from the blinding process (because they are too short), we inspected whether constants actually *survive* constant blinding. Indeed, four-byte gadgets give an adversary more flexibility in the types of gadgets to use and make building a ROP chain significantly easier. But the more fundamental observation is the fact that we found that constants that *should have been* blinded were, in fact, not successfully blinded.

### B. Limitations of Dachshund

DACHSHUND uses code fuzzing, which is known to be incomplete in terms of code coverage and cases that it explores. We have shown that our technique is quite successful for discovering leftover constants in JIT-compiled code. However, similar to other fuzzing techniques, DACHSHUND cannot be used to *prove* that JIT engines do not emit attacker-controlled constants. DACHSHUND could be combined with static code analysis to fulfill this higher goal.

Furthermore, DACHSHUND leverages immediate constants in JavaScript code to evade constant blinding. It may also be possible to find other types of attacker-controllable constants, such as values embedded in control-flow statements (e.g., constants encoded in relative offsets). However, our findings show that an attacker does not even need to search for other types of constants, given plenty of immediate ones.

### C. Limitations of the Defense

Our proposal to defend against constants has certain limitations, which we address next. As already mentioned, our proxy-based solution requires that HTTPS-secured content can also be rewritten. This means that certificate validation will be done by the proxy and the client needs to trust certificates handed out by the proxy. However, in corporate settings, such HTTPS-enabled proxies are quite common and serve to inspect client communication for multiple purposes (e.g., caching, protecting against information leakage, identifying HTTPS-based malware communication, etc.) Our rewriting logic can easily be integrated into existing proxies.

An alternative to a proxy implementation would be to embed our method as a browser extension. This way users will have more control over the rewriter, e.g., they can request on-demand rewriting of certain pages or whitelist trusted pages. Also, HTTPS-based content will be no problem for an extension-based rewriter, as rewriting happens after a page is already loaded. However, the extension will face a race

condition with the running JavaScript, i.e., JavaScript on the page may be executed before the extension finishes rewriting. This problem can be solved by disabling JavaScript execution for all pages until the rewriter terminates.

A technical challenge for our solution is potential deficiencies in the HTML/JavaScript parser. We might face cases in which the parser fails to parse the code. There are two possibilities for dealing with unparsable scripts: (i) we block the script, or (ii) we allow the script without modification. Solution (ii) lowers the security, because an adversary may find out a way to create a script that is unparsable but is still tolerated (and executed) in browsers. However, using (i) may block scripts that come from legitimate sources, thus modifying the semantics of the page. A solution to the aforementioned problem could be to extract all immediate constants by alternative means (e.g., using regular expressions) from unparsable scripts and replace them with the safe alternatives. However, in summary, non-parseable scripts are not a fundamental problem of our approach, but more a technical challenge for parser implementations.

One weakness of our defense technique is that future JIT compilers might convert global objects into integers as part of the JIT code optimization. The basic idea why we replace integer constants with global objects is that the global variables in JavaScript are volatile and can be modified by every function running in the same context, e.g., by a JavaScript function running at some time intervals. Therefore, these global variables, even though they encode integer values, will not be inlined. However, if a global variable is first moved into a local variable, then the local variable can be inlined if necessary. We manually tested multiple such cases in Chrome and found out that the JIT compiler of Chrome does not inline such variables. However, in the future, if the compiler is extended to also inline these variables, our rewriter has to be adapted accordingly.

Furthermore, we unfortunately have no way to prove the completeness of the rewriter. For example, our current prototype implementation does not cover all border cases of implicit conversions. In our JavaScript rewriter, we account for implicit conversions between a string and a number, e.g., from the string `'123'` to the number `123`. JavaScript, however, allows more cases of allowed implicit conversions. For example, using Boolean constants `true` and `false` as numbers `1` and `0` respectively (`true+true` is 2, `true*100` is 100). Similarly, unary operators can be applied to various types of JavaScript objects to convert them to integers. For example, `+[]` equals to 0, and +!![] equals to 1. These types of conversions are used by JavaScript obfuscators to hide the source code [32]. As Edge only caches (i.e., emits in JIT-code) integer constants that are directly encoded as immediate values in JavaScript, these implicit conversions will not be a problem for it. However, they can still be emitted by the optimizing compiler of Chrome. By manual verification, we found out that these cases are *not* optimized by Chrome's current JIT compiler and therefore can be ignored by our defense altogether. In the future, if these values get inlined into the executable code, our defense can be easily extended to also cover them.

Apart from immediate constants, an attacker might encode *implicit* constants in JITted code. She can do so by abusing other parts of the JavaScript code that indirectly influence values encoded in JIT-compiled code. For example, parameters on the stack are referenced by adding their offset to `rbp`. The offset is encoded as a part of the instruction, and thus is emitted to the code. By varying the number of function parameters, the attacker might generate useful gadgets. However, this attack is limited by the number of possible arguments that a function can have, limiting the attacker to incomplete two bytes. Alternatively, Maisuradze *et al.* [36] demonstrated that an adversary can use relative offsets encoded in control flow instructions (e.g., conditional jumps or calls). By carefully choosing certain code sizes, attackers can change the values encoded in these instructions, such as relative offsets of branches (e.g., `if/else`) or calls (e.g., between caller and callee). Complementary techniques, such as code randomization (e.g., NOP insertions) or control-flow-changing code rewriting might help to defend against such cases as a probabilistic defense. We leave these ideas open for future work.

The discussion above has shown that we are not aware of any obfuscation technique that evades our defense. That said, it might be possible that JIT compilers change, or simply that attackers may find novel evasion tricks that we have not discussed. In any case, this is not a fundamental limitation of our defense, but (as the examples above show) we can likely further improve code rewriting to gain complete coverage over any attacker-controlled constant that we might have missed in the current prototype implementation.

## VII. Related Work

In the following, we survey related work, including an evolution of attack techniques and their corresponding defenses.

### A. ASLR vs. Code-Reuse Attacks

ASLR continues to be the most-widely deployed defense against code-reuse attacks [50]. However, apart from being incomplete [46] (i.e., not being applied to all memory segments) or having low entropy due to a 32-bit systems [48], ASLR is also vulnerable to code-reuse attacks utilizing information leakage [31], [17], [5]. To make up for ASLR's weaknesses, fine-grained randomization schemes complement ASLR by randomizing the code *within* memory segments reordered by ASLR. Therefore, leaking a code pointer does not reveal any information about the remaining code in that page. For example, Pappas *et al.* [43] randomize instructions inside basic blocks by code rewriting. ASLP, by Kil *et al.* [33], shuffles the addresses of functions along with important data structures by statically rewriting an executable. STIR, by Wartell *et al.* [51], permutes basic blocks of the program at startup. Lu *et al.* advance these schemes by providing a practical runtime re-randomization solution [35].

However, scripting environments enabled attackers to leverage information leak to bypass ASLR. In JIT-ROP [49], Snow *et al.* demonstrated that by repeatedly exploiting a memory disclosure vulnerability, the attacker can read code pages of a program and generate a gadget chain on the fly.

Closest related to our work, Athanasakis *et al.* [2] empowered JIT-ROP by utilizing the code output from the JIT compilers to *inject* their own gadgets. Knowing that JIT engines do not blind smaller constants, they show that an attacker may be able to carefully align two-byte gadgets to mount successful attacks. We follow the same motivation,

but show that the deficiencies of constant blinding are far more fundamental than ignoring small constants. DACHSHUND has proven that constant blinding implementations in modern browsers are inherently incomplete, irrespective of the size of the constants. In addition, we propose and implement a viable defense against attacker-induced gadgets in JavaScript code.

### B. Defenses against Code Reuse

Researchers have proposed various defenses against code-reuse attacks, as summarized in the following.

**Non-Readable Code:** Backes *et al.* [3] and Crane *et al.* [12] proposed tackling JIT-ROP attacks by forbidding the attacker to read executable pages of the program. XnR (Execute-no-Read) marks executable pages as non-present and utilizes a page-fault handler to allow only valid accesses (i.e., instruction fetches). Similarly, Readactor uses Extended Page Tables (EPT) to mark all executable pages as non-readable and applies fine-grained randomization to all executable pages. Some remaining weaknesses of Readactor (e.g., function pointers in import tables and vtables) have been resolved in its successor Readactor++ [13]. Targeting ARM, also Braden *et al.* [8] suggest to leverage execute-only memory to protect against code-reuse attacks. Finally, Gionta *et al.* suggest to hide code via a split TLB [24].

Although the idea of non-readable code is promising, withdrawing read privileges alone does not suffice to protect against attacker-induced gadgets, in particular if gadgets are deterministic and their locations predictable. This is also the reason why the schemes are typically combined with fine-grained randomization schemes, and hence, their security against our attack heavily depends on the randomization.

**Control Flow Integrity:** CFI schemes restrict the control flow to valid code paths. CFI implementations range from coarse-grained to fine-grained schemes [55], [56], [14], [37], [55], following the typical compromise between efficiency and security [15], [25]. Shying the complexity of JIT engines, few CFI schemes have been tested on JIT compilers. One of the notable exceptions is NaCl SFI [1], which provides a coarse-grained CFI implementation for JIT engines, but faces an overhead of 51% on x64 systems. Similarly, RockJIT instruments JIT-compiled code with coarse-grained checks, verifying the control flow instruction targets at runtime. Forcing the jump targets to be aligned instructions, RockJIT thus successfully defends against our attack. Note that, apart from being fine-grained, the completeness of CFI schemes is equally important, i.e., even in the presence of a single unchecked (or wrongly checked) jump target, the attacker will be able to mount a successful attack. In particular, with arbitrary four-byte gadgets, the attacker only uses unaligned instructions, and therefore no additional CFI checks will be executed in between. Note that this may not be the case for Athanasakis' attack that requires to align multiple shorter gadgets to obtain a useful one. Summarizing, complete CFI schemes are a powerful defense, and may become a viable solution in the long run. However, special attention must be paid to the completeness and to the precision of the sandbox. In the past, sandbox escaping attacks demonstrated that orthogonal defenses, like ours, present a useful additional layer of security.

### C. Protecting JIT Compilers

Next to general code reuse defenses, researchers have also suggested to specifically protect JIT compilers against exploitation. In JITDefender, Chen *et al.* [10] proposed defending against JIT spraying by removing executable rights from JIT-compiled code pages, until they are called by the compiler. Similarly, executable rights will be stripped after the function returned. This way, diverting the control flow to the sprayed code will crash the program. Although this defense may work in some situations, the attacker can extend the time a code pages is executable, e.g., by using a thread that continuously calls a JavaScript function.

Chen *et al.* proposed JITSafe [11]. JITSafe is an extended version of JITDefender, incorporating a similar technique as suggested by Wu *et al.* with RIM [53], to inject invalid instructions into long chains of NOP sleds. While this defense is successful to prevent code spraying with long NOP sleds, it cannot protect against more fine-grained code injections (such as injecting single gadgets, as in our attack).

Homescu *et al.* [29] and Wei *et al.* [52] propose librando and INSeRT, respectively. These techniques are similar to techniques deployed in modern browsers. For example, both of these techniques randomize the JIT-compiled code by randomly inserting either NOP (librando) or illegal (INSeRT) instructions into the code. Moreover, both of these techniques deploy some form of constant blinding, e.g., by using an XOR (INSeRT) or LEA (librando) instruction to encrypt the constants. Our evaluation on popular browsers has proven that such constant blinding schemes are actually hard to get right. To foster future research in this direction, we thus provide DACHSHUND as framework to evaluate the completeness of constant blinding implementations.

### D. JavaScript Rewriting

While with totally different goals in mind, other researchers also used JavaScript rewriting as technique to guarantee various other security aspects. For example, Doupe et al. suggest a Web rewriting framework called deDacota that separates code (JavaScript) from data (HTML) to defend against cross-site scripting (XSS) attacks [16]. Reis et al. rewrite Web documents in such a way that also dynamic contents (e.g., script code) is instrumented and can be validated against security policies [44]. Similarly, Yu et al. provide a provably correct JavaScript code rewriting methodology to defend against threats like XSS [54]. These ideas follow similar concepts to identify JavaScript code in a Web site, however, do not focus on the security of JIT compilers.

## VIII. CONCLUSION

DACHSHUND has uncovered that constant blinding implementations in many popular browsers are incomplete and inherently insecure. This has severe implications on the security of browsers, as (i) the guarantees that are assumed to be given by constant blinding are not met in practice, (ii) we demonstrate how easy an attacker can inject arbitrary gadgets (up to four bytes) to form ROP chains, and (iii) as the problems of constant blinding are far deeper than it was previously believed. Our JavaScript-based rewriting approach is a first step to remove the risk of attacker-induced constants and to

safe the guarantees of constant blinding, without any need to rewrite browser software. In the long run, we presume that more fundamental changes are required to guarantee browser security, such as enforcing Control Flow Integrity schemes even on JIT-compiled code, or exploring provably-secure gadget-free JIT compilers.

## ACKNOWLEDGMENT

The authors would like to thank the anonymous reviewers for their valuable comments. Moreover, we would like to thank Michael Brengel for his feedback during the writing process of the paper.

This work was supported by the German Federal Ministry of Education and Research (BMBF) through funding for the Center for IT-Security, Privacy and Accountability (CISPA) and for the BMBF project 13N13250.

## REFERENCES

[1] 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.

[2] M. Athanasakis, E. Athanasopoulos, M. Polychronakis, G. Portokalidis, and S. Ioannidis, "The Devil is in the Constants: Bypassing Defenses in Browser JIT Engines," in *Proceedings of the Network and Distributed System Security (NDSS) Symposium*, February 2015.

[3] M. Backes, T. Holz, B. Kollenda, P. Koppe, S. Nürnberger, and J. Pewny, "You Can Run but You Can'T Read: Preventing Disclosure Exploits in Executable Code," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '14. New York, NY, USA: ACM, 2014, pp. 1342–1353. [Online]. Available: http://doi.acm.org/10.1145/2660267.2660378

[4] "Baracuda networks." [Online]. Available: https://campus.barracuda.com/product/websecuritygateway/article/BWF/UsingSSLInspection/

[5] A. Bittau, A. Belay, A. Mashtizadeh, D. Mazières, and D. Boneh, "Hacking Blind," in *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, ser. SP '14, Washington, DC, USA, 2014, pp. 227–242.

[6] D. Blazakis, "Interpreter Exploitation," in *Proceedings of the 4th USENIX Conference on Offensive Technologies*, ser. WOOT'10, 2010.

[7] "Blue coat." [Online]. Available: https://www.bluecoat.com/products-and-solutions/ssl-visibility-appliance

[8] K. Braden, S. Crane, L. Davi, M. Franz, P. Larsen, C. Liebchen, and A.-R. Sadeghi, "Leakage-resilient layout randomization for mobile devices," in *23rd Annual Network & Distributed System Security Symposium (NDSS)*, Feb. 2016.

[9] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy, "Return-oriented Programming Without Returns," in *Proceedings of the 17th ACM Conference on Computer and Communications Security*, ser. CCS '10. New York, NY, USA: ACM, 2010, pp. 559–572. [Online]. Available: http://doi.acm.org/10.1145/1866307.1866370

[10] 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*, ser. IFIP Advances in Information and Communication Technology, J. Camenisch, S. Fischer-Hbner, Y. Murayama, A. Portmann, and C. Rieder, Eds. Springer Berlin Heidelberg, 2011, vol. 354, pp. 142–153. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-21424-0_12

[11] 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. [Online]. Available: http://dx.doi.org/10.1049/iet-ifs.2012.0142

[12] S. Crane, C. Liebchen, A. Homescu, L. Davi, P. Larsen, A.-R. Sadeghi, S. Brunthaler, and M. Franz, "Readactor: Practical Code Randomization Resilient to Memory Disclosure," in *36th IEEE Symposium on Security and Privacy (Oakland)*, May 2015.

[13] S. Crane, S. Volckaert, F. Schuster, C. Liebchen, P. Larsen, L. Davi, A.-R. Sadeghi, T. Holz, B. D. Sutter, and M. Franz, "It's a TRAP: Table Randomization and Protection against Function Reuse Attacks," in *Proceedings of 22nd ACM Conference on Computer and Communications Security (CCS)*, 2015.

[14] L. Davi, A. Dmitrienko, M. Egele, T. Fischer, T. Holz, R. Hund, S. Nürnberger, and A.-R. Sadeghi, "Mocfi: A framework to mitigate control-flow attacks on smartphones." in *NDSS*, 2012.

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

[16] A. Doupé, W. Cui, M. H. Jakubowski, M. Peinado, C. Kruegel, and G. Vigna, "dedacota: toward preventing server-side xss via automatic code and data separation," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM, 2013, pp. 1205–1216.

[17] T. Durden, "Bypassing PaX ASLR protection." [Online]. Available: http://phrack.org/issues/59/9.html

[18] "Escodegen: Ecmascript code generator from mozilla's parser api ast." [Online]. Available: https://github.com/estools/escodegen

[19] "Esprima: Ecmascript parsing infrastructure for multipurpose analysis." [Online]. Available: http://esprima.org/

[20] "Estraverse: Ecmascript traversal functions from esmangle project." [Online]. Available: https://github.com/estools/estraverse

[21] "Forcepoint." [Online]. Available: https://www.websense.com/content/support/library/web/v81/wcg_help/ssl_enable.aspx

[22] "Forefront threat management gateway." [Online]. Available: https://technet.microsoft.com/en-us/library/dd441073.aspx

[23] "Fortigate." [Online]. Available: http://cookbook.fortinet.com/why-you-should-use-ssl-inspection/

[24] J. Gionta, W. Enck, and P. Ning, "Hidem: Protecting the contents of userspace memory in the face of disclosure vulnerabilities," in *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy*, ser. CODASPY '15. New York, NY, USA: ACM, 2015, pp. 325–336. [Online]. Available: http://doi.acm.org/10.1145/2699026.2699107

[25] E. Goktas, E. Athanasopoulos, H. Bos, and G. Portokalidis, "Out of control: Overcoming control-flow integrity," in *Security and Privacy (SP), 2014 IEEE Symposium on*. IEEE, 2014, pp. 575–589.

[26] C. Grier, L. Ballard, J. Caballero, N. Chachra, C. J. Dietrich, K. Levchenko, P. Mavrommatis, D. McCoy, A. Nappa, A. Pitsillidis, N. Provos, M. Z. Rafique, M. A. Rajab, C. Rossow, K. Thomas, V. Paxson, S. Savage, and G. M. Voelker, "Manufacturing compromise: The emergence of exploit-as-a-service," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, ser. CCS '12. New York, NY, USA: ACM, 2012, pp. 821–832. [Online]. Available: http://doi.acm.org/10.1145/2382196.2382283

[27] J. Hiser, A. Nguyen-Tuong, M. Co, M. Hall, and J. W. Davidson, "ILR: Where'D My Gadgets Go?" in *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, ser. SP '12, Washington, DC, USA, 2012, pp. 571–585. [Online]. Available: http://dx.doi.org/10.1109/SP.2012.39

[28] C. Holler, K. Herzig, and A. Zeller, "Fuzzing with code fragments," in *Proceedings of the 21st USENIX Conference on Security Symposium*, Berkeley, CA, USA, 2012.

[29] 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*, ser. CCS '13, 2013.

[30] "Http mitm proxy." [Online]. Available: https://github.com/joeferner/node-http-mitm-proxy

[31] a. huku, "Exploiting VLC. A Case Study on Jemalloc Heap Overflows." [Online]. Available: http://www.phrack.org/issues/68/13.html

[32] "Javascript obfuscation." [Online]. Available: http://patriciopalladino.com/blog/2012/08/09/non-alphanumeric-javascript.html

[33] C. Kil, J. Jun, C. Bookholt, J. Xu, and P. Ning, "Address Space Layout Permutation (ASLP): Towards Fine-Grained Randomization of Commodity Software," in *Proceedings of the 22Nd Annual Computer Security Applications Conference*, ser. ACSAC '06, Washington, DC, 2006. [Online]. Available: http://dx.doi.org/10.1109/ACSAC.2006.9

[34] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz, "SoK: Automated Software Diversity," in *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, ser. SP '14, Washington, DC, USA, 2014, pp. 276–291. [Online]. Available: http://dx.doi.org/10.1109/SP.2014.25

[35] K. Lu, S. Nrnberger, M. Backes, and W. Lee, "How to make aslr win the clone wars: Runtime re-randomization," in *Network and Distributed System Security Symposium. Symposium on Network and Distributed System Security (NDSS)*, K. Lu, S. Nrnberger, M. Backes, and W. Lee, Eds. Internet Society, 2015.

[36] G. Maisuradze, M. Backes, and C. Rossow, "What cannot be read, cannot be leveraged? revisiting assumptions of jit-rop defenses," in *25th USENIX Security Symposium (USENIX Security 16)*. Austin, TX: USENIX Association, Aug. 2016, pp. 139–156. [Online]. Available: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/maisuradze

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

[38] Nergal. The Advanced Return-into-lib(c) Exploits. [Online]. Available: http://phrack.org/issues/58/4.html

[39] "Node.js: A javascript runtime built on chrome's v8 javascript engine." [Online]. Available: https://nodejs.org/

[40] "Octane: The javascript benchmark suite for the modern web." [Online]. Available: https://developers.google.com/octane/

[41] K. Onarlioglu, L. Bilge, A. Lanzi, D. Balzarotti, and E. Kirda, "G-Free: Defeating Return-oriented Programming Through Gadget-less Binaries," in *Proceedings of the 26th Annual Computer Security Applications Conference*, ser. ACSAC '10. New York, NY, USA: ACM, 2010, pp. 49–58. [Online]. Available: http://doi.acm.org/10.1145/1920261.1920269

[42] "Palo alto networks." [Online]. Available: https://www.paloaltonetworks.com/documentation/60/pan-os/pan-os/decryption

[43] V. Pappas, M. Polychronakis, and A. D. Keromytis, "Smashing the Gadgets: Hindering Return-Oriented Programming Using In-place Code Randomization," in *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, ser. SP '12, Washington, DC, USA, 2012. [Online]. Available: http://dx.doi.org/10.1109/SP.2012.41

[44] C. Reis, J. Dunagan, H. J. Wang, O. Dubrovsky, and S. Esmeir, "Browsershield: Vulnerability-driven filtering of dynamic html," *ACM Transactions on the Web (TWEB)*, vol. 1, no. 3, p. 11, 2007.

[45] J. Ruderman, "Introducing jsfunfuzz." [Online]. Available: http://www.squarefree.com/2007/08/02/introducing-jsfunfuzz/

[46] E. J. Schwartz, T. Avgerinos, and D. Brumley, "Q: Exploit Hardening Made Easy," in *Proceedings of the 20th USENIX Conference on Security*, ser. SEC'11. Berkeley, CA, USA: USENIX Association, 2011, pp. 25–25. [Online]. Available: http://dl.acm.org/citation.cfm?id=2028067.2028092

[47] H. Shacham, "The Geometry of Innocent Flesh on the Bone: Return-into-libc Without Function Calls (on the x86)," in *Proceedings of the 14th ACM Conference on Computer and Communications Security*, ser. CCS '07. New York, NY, USA: ACM, 2007, pp. 552–561. [Online]. Available: http://doi.acm.org/10.1145/1315245.1315313

[48] 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*, ser. CCS '04. New York, NY, USA: ACM, 2004, pp. 298–307. [Online]. Available: http://doi.acm.org/10.1145/1030083.1030124

[49] 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 *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, ser. SP '13, Washington, DC, USA, 2013, pp. 574–588. [Online]. Available: http://dx.doi.org/10.1109/SP.2013.45

[50] P. Team, "Address Space Layout Randomization (ASLR)." [Online]. Available: http://pax.grsecurity.net/docs/aslr.txt

[51] 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*, ser. CCS '12. New York, NY, USA: ACM, 2012, pp. 157–168. [Online]. Available: http://doi.acm.org/10.1145/2382196.2382216

[52] 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*, March 2011, pp. 323–328.

[53] R. Wu, P. Chen, B. Mao, and L. Xie, "RIM: A Method to Defend from JIT Spraying Attack," in *Proceedings of the 2012 Seventh International Conference on Availability, Reliability and Security*, ser. ARES '12, Washington, DC, USA, 2012, pp. 143–148. [Online]. Available: http://dx.doi.org/10.1109/ARES.2012.11

[54] D. Yu, A. Chander, N. Islam, and I. Serikov, "Javascript instrumentation for browser security," in *ACM SIGPLAN Notices*, vol. 42, no. 1. ACM, 2007, pp. 237–249.

[55] 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.

[56] M. Zhang and R. Sekar, "Control flow integrity for cots binaries." in *Usenix Security*, vol. 13, 2013.

[57] "Zscaler." [Online]. Available: https://www.zscaler.com/products/ssl-inspection