libmask: Protecting Browser JIT Engines from the Devil in the Constants

Abhinav Jangda, Mohit Mishra, Benoit Baudry

libmask: Protecting Browser JIT Engines from the Devil in the Constants

Abhinav*, Mohit Mishra*†, Benoit Baudry‡
*Indian Institute of Technology (BHU) Varanasi
{abhinav.student.apm11, mohit.mishra.cse11}@iitbhu.ac.in
†BrightEdge Technologies, Inc.
mmishra@brightedge.com
‡INRIA Rennes, France.
benoit.baudry@inria.fr

***Abstract*—JavaScript (JS) engines are virtual machines that execute JavaScript code. These engines find frequent application in web browsers like Google Chrome, Mozilla Firefox, Microsoft Internet Explorer and Apple Safari. Since, the purpose of a JS engine is to produce executable code, it cannot be run in a non-executable environment, and is susceptible to attacks like Just-in-Time (JIT) Spraying, which embed return-oriented programming (ROP) gadgets in arithmetic or logical instructions as immediate offsets. This paper introduces `libmask`, a JIT compiler extension to prevent the JIT-spraying attacks as an effective alternative to XOR based constant blinding. `libmask` transforms constants into global variables and marks the memory area for these global variables as read only. Hence, any constant is referred to by a memory address making exploitation of arithmetic and logical instructions more difficult. Further, these memory addresses are randomized to further harden the security. The scheme has been implemented and evaluated as a librddy extension to Google V8 scripting engine with optimizations that contain performance overhead and make `libmask` a feasible approach. We demonstrate that `libmask` masks all the constants in JITed code, and effectively raise the bar for JIT-spray and JIT-ROP attacks. The average overhead incurred upon memory is less than 300 kilobytes, while in most benchmarks the memory overhead is less than 10 KB. The average performance overhead observed with optimizations measures is 5.31%. Further, this new approach shows a modest performance improvement over currently deployed constant blinding technique in Google V8.**

I. Introduction

At the heart of web browsers lie JavaScript engines - virtual machines that execute the JavaScript code. Some of the popular JavaScript engines are: Google V8 (in Google Chrome)[30], SpiderMonkey (in Mozilla Firefox)[31], Chakra (in Microsoft IE and Edge)[32], JavaScriptCore (in Apple Safari)[33], and Mozilla's Rhino (completely written in Java)[34]. With such proliferation of web browsers in the market, it becomes equally important to harden them against security vulnerabilities and exploits. Attackers often abuse vulnerabilities and bugs to conduct a successful exploit on the system or application. In response to these exploits, multiple defense mechanisms have been proposed and deployed in modern browsers. Data Execution Prevention (DEP)[4] is widely deployed across browsers and systems. DEP is a technique that prevents input data (from the attacker) from being executed, thus preventing payload injection attacks. However, Shacham et al. [18] showed that DEP can be bypassed, using a technique called Return-Oriented Programming (ROP)[18]. ROP is an effective code-reuse attack that exploits a set of Turing-complete code snippets called ROP gadgets in the program and chain them together to deploy an attack.

JIT-spraying attacks allow an attacker to bypass data execution prevention (DEP) and memory randomization (ASLR). For this, the attacker embeds a ROP gadget as an immediate offset in XOR instructions. Subsequently, the attacker forces the JIT-compiler to emit many of these malicious instructions so that the attacker's chance of guessing the location of one of those instructions is high.

This paper introduces `libmask`, a non-XORing constant blinding scheme, to prevent the so-called JIT-spraying attacks. These attacks embed ROP gadgets in arithmetic and/or logical instructions as immediate offsets. The motivation behind this scheme is to reduce the performance overhead and increase security of the currently deployed XOR based constant blinding. For example, in case of Google V8, only integer constant of size greater than 4 bytes are blinded. A constant is first XORed with a random key to generate the obfuscated constant. The constants is XORed back again when the original constant needs to be used. Athanasakis et al. in [17] has shown that XOR constant blinding technique can be bypassed. Their technique contains constants, which are less than 4 bytes and hence, are emitted as such in the JIT code. To mitigate this issue, we replace a constant with a randomized address.

The fundamental idea is to replace each constant with a random address. This random address stores the value of the constant. This is achieved by replacing constants with a global variable in the source code. All constants within the source code are allocated at random addresses by random shuffling and adding random padding. The memory area of these random addresses is then marked as read-only. We then present several optimizations to improve the performance. These optimizations includes Register Allocation and Constant Caching to improve the performance of constant masked code and masking constants in parallel to decrease the JIT compilation overhead.

We evaluated `libmask` against XOR based constant blinding in Google V8 and discovered that `libmask` performs better than XOR based constant blinding both in terms of

security and performance.

a) Contributions: This paper makes the following contributions:

- We introduce the design of `libmask`, a JIT Compiler extension as an alternative to XOR based constant blinding (Section III).
- We present various optimizations for `libmask` to improve the performance of JIT code and reduce JIT compiler overhead (Section IV).
- We plugged in `libmask` into Google V8 and performed extensive evaluations on Octane Benchmarks [16] (Section V). Our evaluations show that:
  - The average performance overhead of `libmask` is 5.31%.
  - The average memory requirement of `libmask` is less than 300 KB, while a number of benchmarks show an average requirement of just 10 KB.
  - `libmask` performs better than XOR based constant blinding extensively deployed in modern browsers like Google Chrome.

II. Background

A. Just-In-Time Compiler

A Just-in-time (JIT) compiler generates native code during the execution of the program. JIT compilation combines two traditional approaches: ahead of time compilation and interpretation. JIT compilers emit native code during program run-time. Like interpreters, they translate bytecode to machine code continuously but employ caching of translated code to avoid re-translating the previously translated code. Several modern run-time environments employ JIT compilation techniques like most implementations of Java, Microsoft .NET Framework, or Google's V8 Javascript Engine.

However, the JITed code is predictable in nature. In actual practice, extremely minimal variations are introduced in the JITed code: the JIT compiler always emits out the same native code for a certain piece of source code. This predictability nature puts its security under question since adversaries can predict the native code with accuracy.

B. ROP and JIT Spray

Return-oriented programming is a non code-injection attack technique. It differs from the classical return-to-libc[18] in the way that it utilizes small instruction sequences (ending with `ret` instruction) present in either the binary or libraries linked to the application. These instruction sequences are called gadgets. The following example illustrates how ROP attack works.

Consider loading a constant into a register, which will allow to save the value in the stack, and enabling it to be popped when required. The following instruction would pop the value from the stack into the EAX register and return the address on the stack's top, which can be the address to the next gadget.

```
1  pop eax;
2  ret;
```

pop eax; ret;
0x00dead;
next gadget's address;

Fig. 1. Stack representation of the gadget

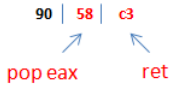

Fig. 2. Intended Gadget - Loading a constant in a register and popping it when required.

The stack diagram in Figure 1 explains how the above gadget helps to return to the next gadget, which further can be chained to perform a successful exploit.

JIT Spray[5], a form of ROP attack exploits a JIT Compiler's behavior and bypasses Address Space Layout Randomization [18] and DEP [4]. As mentioned earlier, JIT compilers are predictable in the way that the JITed code shows minimal variations across multiple compilations. Also, a JIT compiler generates executable code, and hence it must run in an executable environment. Therefore, a JIT compiler is usually exempt from DEP. A JIT Spray attack works by performing heap spraying[36] with the generated native code that facilitates arbitrary code execution. In case of web browsers, heap spraying makes use of the JavaScript code and spray the heap by creating large strings, followed by making copies of the long string with a shellcode. These copies are stored in an array, till the point where sufficient memory has been sprayed. This ensures that the exploit would work.

C. Constant Blinding/Masking

There are two ways a constant in the source code could land up in the executable code: (i) as a value stored by the compiler close to the code (without executing it as code), or (ii) as an immediate instruction operand.

For the first case, it suffices to add padding, e.g. NOP insertion, randomizing the address location of the value, making it hard to guess for the adversaries. For the later case, one of the currently deployed defenses in modern browsers effective against JIT Spray attacks is constant blinding. This involves a generated random key (or cookie) used to XOR an immediate operand, and then XORed again when it is actually used. For instance, assume the following sample JavaScript code:

```
1  var a = 0x9058C3
```

The native code (without any constant blinding mechanism) would include the constant as it is. Note that this constant can potentially be exploited since, it can act as an intended gadget.

Figure 2 shows that the combination of `58` and `c3` can act as gadget, which can pop a value as and when required from the stack. To prevent this to happen, modern web browsers deploy constant blinding, wherein the constant is XORed with a random key (called cookie) and XORed again when the constant needs to be used. This way, the constant never ends up as it is in the JIT buffer, raising the bar for the adversary

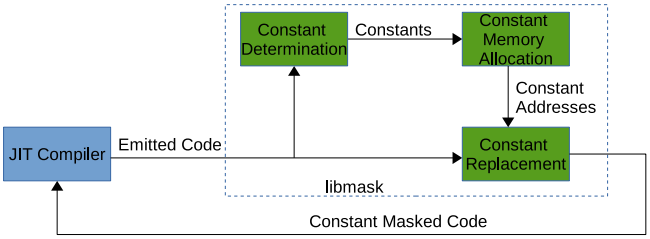

Fig. 3. Design and working of libmask

to predict the value without knowing the cookie's value. The variable `a` is never showed up as 0x9058c3, but as 0x826c95 with cookie's value used being 0x123456. Now, when variable `b` wants to use the original value of variable `a`, `a` is XORed again with the same cookie in its operation.

```
1  a = 0x9058c3 ^ 0x123456   //  = 826c95
2  ...
3  b = a ^ 0x123456
```

The machine code generated to implement constant blinding for above example is given below:

```
1  mov rax, 0x826c95  //move XORed constant to
       register
2  xor rax, 0x123456  //xor with cookie
```

As can be observed from above, it becomes hard to guess the original constant's value, since that is never showed up in the native code as it is. However, there are two operations required to obfuscate and use the value of the constant. The current scheme takes care of integer constants of size more than 4 bytes only (e.g. in Google V8). Gadgets can be formed from integers of less than 4 bytes also [17].

III. Design

`libmask` performs constant masking in the code generated by a JIT compiler. It reads the code emitted by JIT compiler, disassembles it, performs constant masking, assembles the constant-masked code (CMC), and returns the CMC to the JIT compiler. This CMC is then executed by the virtual machine. Figure 3 shows the design and working of `libmask`. Currently, `libmask` supports x86-64 architecture, and hence our discussion will stick around with x86-64 architecture in this paper.

Constant masking via `libmask` is divided into three phases: (i) Constant Determination, (ii) Constant Memory Allocation, and (iii) Constant Replacement. *Constant Determination* phase determines all the constants in the machine code. *Constant Memory Allocation* allocates the memory for the constants (constant memory) and marks it as read-only after shuffling and padding. *Constant Replacement* phase replaces each constant found in the machine code with it's respective address in constant memory.

A. Constant Determination

After disassembling the code generated by JIT compiler, this phase traverses each machine instruction inside the code. If any one of the operands to a machine instruction is a constant, then the constant and the position of the machine instruction are recorded.

B. Constant Memory Allocation

After traversing the machine code, the set of all constants is shuffled. Further, random padding is added after each constant. This ensures that the constants are allocated at randomized addresses. After allocating memory, the storage slot corresponding to each constant is initialized with the value of the constant. Finally, this constant memory (memory holding up the constants) area is marked as read-only.

Algorithm 1 Constant Memory Allocation Algorithm

```
1:  random_shuffle (constantVector)
2:  posArray ← new int [constantVector.length()+1]
3:  posArray[1] ← 0
4:  for all i = 2 → constantVector.length()+1 do
5:      posArray[i] = posArray[i − 1] + sizeof (constantVec-
    tor[i − 1])
6:  for all i = 1 → constantVector.length()+1 do
7:      j ← (rand() % 2) × m
8:      for all k = i → constantVector.length()+1 do
9:          posArray[k] ← posArray[k] + j
10: fix alignment for each constant in posArray
11: memorySize ← posArray[constantVector.length () + 1]
12: constantMemory ← malloc (memorySize)
13: for all i = 1 → constantVector.length () do
14:     memcpy (constantMemory, posArray[i],
    possArray[i+1]-1, constantVector[i])
15: mark constant memory as read-only
```

Algorithm 1 presents constant memory allocation algorithm. It takes two arguments: (i) `constantVector`, which is a vector of constants as an argument and (ii) `m`, which is the length of each padding. Line 1 randomly shuffles `constantVector`. Line 2 creates `posArray`, which contains the position (in bytes) in constant memory for each constant. Lines 3 - 5 set the initial position for each constant based on the size of the previous constant in the array. Lines 6 - 9 add a random padding of atmost `m` bytes after every constant. Line 10 adds the required alignment for every constant and updates the `posArray`. Lines 10 - 11 allocate the constant memory, initialize the memory locations with the constant, and mark the constant memory as read-only.

C. Constant Replacement

After *Constant Memory Generation* phase, *Constant Replacement* will replace constant in machine instruction with its address in the constant memory. In x86-64 architecture, one of the operand in instructions can also be an address. For example, the `add` instruction, can take a constant, register, or a memory location as one of its operand. This memory location will be first de-referenced by the CPU to obtain the operand value. This phase rewrites the instruction in the machine code with the address of constant in constant memory.

D. Example

In this section, we present an example of the working of our technique. For this example we assume an x86 32-bit

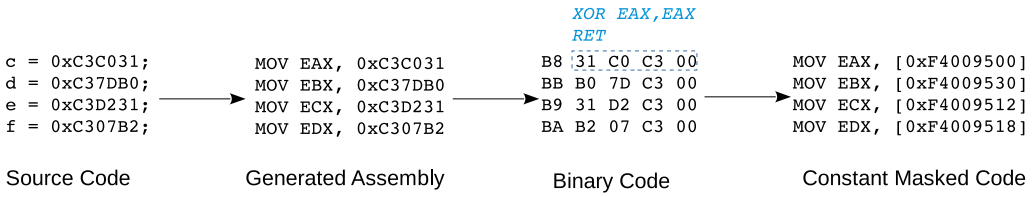

Fig. 4. Example

architecture for simplicity and understanding. Figure 4 shows an example of constant masking using `libmask` in the source code. The source code contains five constants. These constants are the gadgets required for calling `mprotect` [17]. The JITed contains these constants, and hence these constants can be exposed by an attacker. On the other hand, the constant masked code generated by `libmask` replaces these constants with random addresses. These random addresses are accessed by the CPU to obtain the constant. The random addresses are different for every execution of the source program.

IV. Optimizations

`libmask` presented in Section III could result in high performance overhead as given in Section V. This performance overhead is due to the following reasons:

1) Low performance of CMC: An immediate value in the original machine code is replaced by an address to produce the CMC. Since, a memory access is a costly operation compared to using an immediate operand, CMC suffers from this bottleneck.
2) Time taken in diversification of code: The whole process of diversification takes time proportional to the number of constants to be blinded. In Section V, `MandreelLatency` benchmark suffers the highest performance overhead. `MandreelLatency` benchmarks measures the JIT compiler latency.

To decrease the performance overhead due to the above mentioned issues, the following optimizations are performed.

A. Parallel Division of the Machine Code

Optimizations presented in Section IV-B and Section IV-C require the JITed code to be divided into loop-free subsequences. A loop-free subsequence is defined as a machine code instruction sequence, which does not contain a loop i.e. a loop-free subsequence does not contain a backward branch. Every backward branch divides the code into two loop-free subsequences: (i) one between the previous loop-free subsequence and the target of the backward branch, (ii) and the other between the target of the backward branch and the backward branch itself.

In this section, we present a parallel algorithm for dividing a machine code into loop-free subsequences. To divide the machine code into loop-free subsequences in parallel, different threads are created at first. The number of threads is set to be equal to the number of CPU cores. The machine code is then divided equally to each of the threads. Each thread then examines each instruction individually. If an instruction is a backward branch, then the target of the backward branch is recorded into the thread's local database. Finally, each thread's local database is merged serially to the main thread database.

B. Register Allocation

Main Memory access is a costly operation as compared to access from register or using immediate operands. Accessing same memory address more than once is costlier than accessing the memory address once, storing the value in a register, and accessing that register when required. Therefore, we developed our `Register Allocation` optimization algorithm to decrease the memory access overhead.

Initially, the constant masked code is divided into loop-free subsequences as given in Section IV-A. In a loop-free subsequence, for each register *r* all parts of a loop-free subsequence between each access of *r* is determined. Let us call each of these parts of a loop-free subsequence as *access range* for a register *r*. Hence, in each of the *access range* of register *r*, there are no accesses to the *r* i.e. for each instruction in *access range* of *r*, *r* is neither used as a source operand or a destination operand. In each of the *access range* for a register *r*, the occurrence of each constant memory address as an operand to any instruction is counted. In an *access range* of register *r*, for a constant memory address with maximum number of accesses, `push` *r* and `mov` *r*, `[<constant memory address>]` instructions are inserted before the first instruction of *access range* for *r*, each occurrence of the constant memory address is replaced by *r*, and `pop` *r* instruction is inserted after the last instruction of *address range* of *r*. This process is then done for all access range of all registers. Note that `push`, `pop`, and `mov` instructions do not change the flag register.

If the *access range* for a register *r* is the loop, then above technique also has the advantage of bringing the `push` and `mov` instructions outside of the loop.

C. Constant Caching

Since, constants have to be loaded from the constants memory, the CPU can take advantage of the memory locality. To improve the locality provided by the technique in Section III, we develop Constant Caching optimization.

In a loop-free subsequence, if one constant is accessed then the CPU can take the advantage of the memory locality, to also load other constants of a loop-free subsequence in the cache. To acheive this, instead of shuffling all constants in the constant un-masked code, Constant Caching optimization

first shuffles the constants of a loop-free subsequence and then shuffles each group of constants of a loop-free subsequence.

Alogrithm 2 represents the Constant Caching Algorithm. Lines 1 - 2 shuffle constants for each loop-free subsequence. Line 3 shuffles each loop-free subsequence. To add Constant Caching in our technique, Line 1 of Algorithm 1 is replaced by Alogrithm 2.

Algorithm 2 Constant Caching Algorithm

```
1: for all seq ∈ loopfreeSubSequences do
2:     random_shuffle (constantVector, seq.start, seq.end)
3: random_shuffle_subsequence (constantVector)
```

D. Parallel Constant Determination

The *Constant Determination* phase given in III-A is serial. Hence, *Constant Determination* phase would perform poorly on benchmarks with huge number of constants. To solve this issue, we developed *Parallel Constant Determination* optimization. In this optimization, the disassembled code generated by the JIT compiler is divided equally to different threads. The number of threads is set equal to the number of CPU cores. Each thread then examines each instruction. If the operand to any instruction is a constant, it is recorded in a threads local database. Each thread's local database is then merged serially to the main thread database.

V. Evaluation

In this section, we provide our implementation of `libmask`, evaluation techniques, environment, and results. Our evaluation is based on the performance, memory requirement, and security impact.

A. Implementation

Google Chrome occupies over 60% [15] of the market share of web browsers. Hence, it was imperative to implement and evaluate our technique on Google V8. Like most virtual machines, Google V8 has a Garbage Collector and a Just-In-Time compiler. However, V8 does not use any byte code representation. Instead, it directly converts JavaScript code to machine code.

B. Environment

We evaluated our implementation on a 4 core 2.30 GHz Intel Core i7-3610QM machine with 6 GB of RAM running Fedora 21 with Linux Kernel version 3.19 and executing Octane 2.0 JavaScript Benchmarks [16].

C. Performance Evaluation

We performed performance evaluation on different configurations. In each of these configurations, we fixed `m` = 8, which is the length of each padding in Algorithm 1. The configurations are:

- *V8* configuration is Google V8 without `libmask` and XOR based constant blinding disabled.
- *`libmask`* configuration is Google V8 with `libmask` presented in Section III without any optimization.
- *`libmask`+O* configuration is Google V8 integrated with `libmask` with all optimizations enabled.
- *XOR* is Google V8 with XOR based constant blinding enabled. Google V8 implements a XOR based Constant Blinding by XORing an integer of size more than 4 bytes with a random key and then XORing it again when it is actually used.
- *`libmask`+O*+4b configuration is Google V8 integrated with `libmask` with all optimizations enabled but only constants of greater than 4 bytes are masked. This configuration is used to compare with *XOR* configuration.

Each benchmark was executed five times for each of the configurations and average time was calculated. Figure 5 shows the performance overhead for *`libmask`* over *V8*, *`libmask`+O* over *V8*, and *XOR* over *`libmask`+O*+4b. Observe that with the optimizations enabled the overhead of `libmask` has significantly decreased. `MandreelLatency` benchmark, which measures compiler latency shows significant performance overhead of 50.75% with *`libmask`* configuration. This is because `Mandreel` benchmark consists of over 19,000 integer constants and blinding all of them incurs noticeable overhead. However, with optimizations enabled *`libmask`*, this overhead reduces to 19% because the constants are now being masked in parallel. In `Mandreel` benchmark which shows the highest overhead, there are large number of constants in the functions with high amount of hotness. On the other hand, for the benchmarks with low overhead like `Richards` the number of constants in the functions with high amount of hotness are very less. The average performance overhead observed with optimizations enabled is 5.31%. Figure 5 also shows that *XOR* configuration has some performance overhead over *`libmask`+O+4b*. This performance overhead can be attributed to the following reasons:

- XOR based constant blinding adds 2 instructions in the machine code, while *`libmask`+O+4b* adds only 1 instruction.
- Memory addressing is a costly operation but with *Constant Caching* and *Register Allocation* optimizations, instead of memory access either register access or cache access is used to obtain the constant.
- XOR based constant blinding does not implement techniques to decrease the JIT compilation latency. With `libmask` working over the source code in parallel, there is a significant decrease in the overhead.

D. Memory Requirements

Table I shows the number of integer constants and size of constant memory in KBs for each benchmark. The size of constant memory is directly proportional to the number of constants. Hence, `Mandreel` benchmark with 274842 constants requires 3 MB for constant memory and `zlib` with lowest number of constant requires only 0.7 KB for constant memory. On an average, the size of constant memory required is only 295.7 KBs.

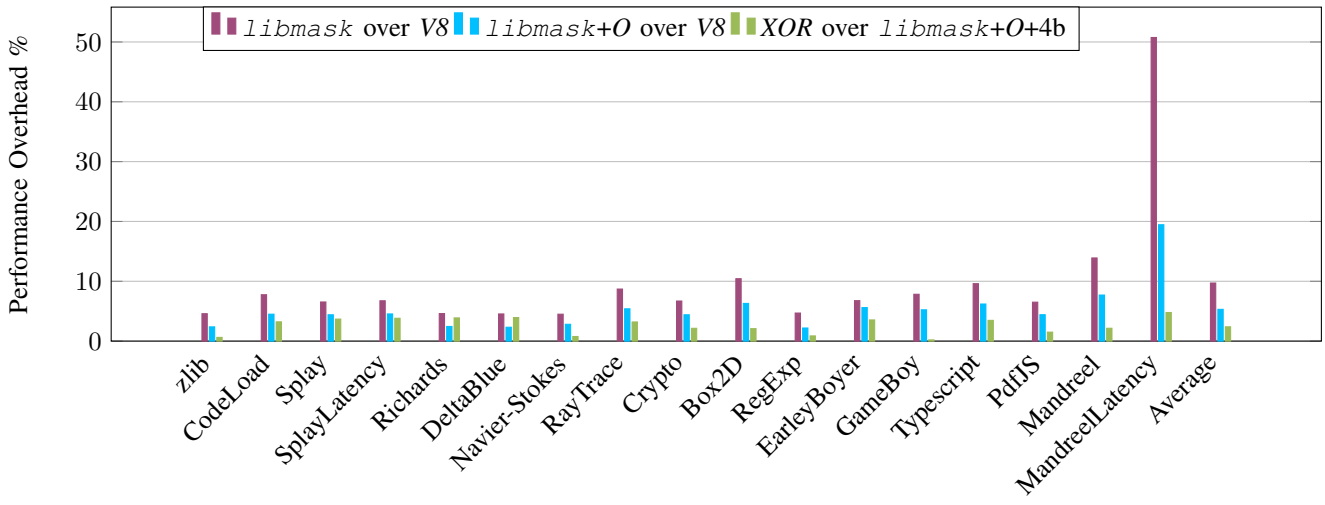

Fig. 5. Performance Overhead (in %) for different configurations

Benchmark	# of Integer Constants	Constant Memory Size (in KBs)
Richards	98	1.1
DeltaBlue	115	1.3
Crypto	800	9.6
RayTrace	176	2.1
EarleyBoyer	555	6.6
Splay	98	1.1
GameBoy	6211	74.5
CodeLoad	72	0.86
Mandreel	274842	3298.1
zlib	59	0.7
Typescript	4276	51.3
PdfJS	80446	965.3
RegExp	312	3.7
Navier-Stokes	199	2.3
Box2D	1475	17.7
Average	24648	295.7

TABLE I
NUMBER OF INTEGER CONSTANTS AND CONSTANT MEMORY SIZE (IN KBS) FOR OCTANE BENCHMARKS

E. Security Evaluation

We performed security evaluation of `libmask` using two approaches: (i) security analysis and (ii) implementing real world attacks.

1) Security Analysis: The security provided by `libmask` is based on the fact that constants appearing in the source code are replaced by random memory addresses. Hence, the attacker is not able to exploit the original constants any longer, since, they completely get masked under randomized addresses and are not available in the JITed code. Moreover, in each run of the program the constants are masked to different addresses, i.e. no constant gets the same memory address across different runs of the program, thereby significantly reducing any chance for attacker to guess the memory address. Since, these are random addresses, it is highly improbable that the attacker could get a useful gadget.

Assuming there are n constants in the source program, there are $n!$ ways to arrange constants in the constant memory and there are 2^n different ways to add padding in the constant memory. Hence, for `n` constants there are $2^n \times n!$ different addresses. Table II depicts the number of possible array layouts for different values of `n`. With `n` = 32, there are 1.1×10^{45} possible memory addresses. For `n` = 16, there are 1.3×10^{18} possible memory addresses. Therefore, the probability of guessing the correct array layout is far too low to consider to be feasible. For `n` = 16, the probability is 7.7×10^{-19}.

We chose two attacks presented in [17] for evaluating the security provided by `libmask`. One JavaScript attack is shown in Figure 6. This attack contains 10 distinct constants, hence there are $2^{10} \times 10! = 3715891200$ possible addresses, which can mask the constants. The second JavaScript attack is shown in Figure 7. This attack contains 7 distinct constants, hence, there are $2^7 \times 7! = 645120$ possible addresses, which can mask the constants. We executed both these attacks on V8 with `libmask` and all the optimizations enabled. We found that constants did not appear as such in the instructions, instead they were masked by random addresses. We further found that everytime we execute the attacks on V8 with `libmask` and all the optimizations enabled, the disassembly of these random addresses is different from what is required to produce the attack. No gadget chaining could be then performed to allow arbitrary code execution.

VI. RELATED WORK

A lot of work has been done to develop defenses against code re-use attacks. One of the first ideas for raising the bar of attack by randomization was presented in [1]. Several ways of achieving this like substituting equivalent instructions, instruction reordering, extra jump or call insertions and many others were introduced by Cohen [1]. These ideas were further enhanced in [2].

Code Randomization has been one of the most common defense against code re-use attacks. Randomization techniques for achieving code randomization includes instruction and

Number of Constants (n)	Cases for inserting padding (2^n)	Constant Arrangments (n!)	Possible Memory Addresses ($2^n \times n!$)
2	2^2	2!	8
4	2^4	4!	384
8	2^8	8!	10^7
16	2^{16}	16!	1.3×10^{18}
32	2^{32}	32!	1.1×10^{45}

TABLE II
POSSIBLE CONSTANT MEMORY ADDRESSES FOR DIFFERENT NUMBER OF CONSTANTS

```
var g1 = 0;
...
var g7 = 0;

for (var i = 0; i < 100000; i++)
{
  g1 = 50011; // pop ebx; ret;
    g2 = 50009; // pop ecx; ret;
    g3 = 12828721; // xor eax, eax; ret;
    g4 = 12811696; // mov 0x7d, al; ret;
    g5 = 12833329; // xor edx, edx; ret;
    g6 = 12781490; // mov 0x7, dl; ret;
    g7 = 12812493; // int 0x80; ret;
}
```

Fig. 6. Attack 1

```
function r8(addr)
{
  return addr + 0x5841;
}

function r9(addr)
{
  return addr + 0x5941;
}

function emitgadgets ()
{
  for (i = 0; i < 0xc35841;  i++)
  {
    rax = 0xc358;
    rcx = 0xc359;
    rdx = 0xc35a;
    r8 (0);
    r9 (0);
  }

  return 0;
}

emitgadgets ();
```

Fig. 7. Attack 2

basic block reordering, register re-allocation, instruction substitution and NOP insertion. These techniques are implemented by [3], [6], [7] in compiler. [9], [8] and [29] implements these techniques in virtual machines and JIT. [10], [11] uses static binary re-writing to implement above techniques. [8] applies above techniques on dynamically generated code. These implementations show that diversification has a positive impact on software security with negligible performance overhead. [8] relies on static NOP insertion techniques while [29] has shown that even less performance overhead could be achieved using dynamic adaptive NOP insertion.

One of the most common defenses to achieve constant blinding is to XOR the constant with a randomly generated key (cookie) and XOR the obtained value again with the key whenever constants are used. We compared our approach with this technique and found that our approach is significantly better than this approach both in security and in performance. The Chakra JS engine used in Internet Explorer as its JIT engine employs several defenses like NOP insertion and XOR based Constant Blinding. [17] has shown that even if JIT uses the above approach as defense an attacker can still successfully attack the system by generating some specific gadgets.

Song et al. showed in [12] that JIT buffers can be exploited with the help of code injection techniques if the buffer is both writable and executable or even temporarily writable at times. A multi-threaded generated code is more prone to above attack because the switch between writable and executable leaves a time window for the application of exploit. To avoid such exploits they also proposes a dynamic code generation architecture which utilizes a separate process and shared memory.

Address Space Layout Randomization (ASLR) is an operating system level randomization technique. It places significant areas of the program like stack and memory obtained from `mmap` at random addresses. This technique is used by all major operating systems (Windows, Linux and Mac OS). Although this technique has several benefits it is not a sufficient security measure against code reuse attacks. [18] has shown that ASLR has very low entropy and it would take just a few minutes to find the base address. DEP [4] and SafeSEH [28] are another widely adopted lightweight protection mechanism. However, like ASLR these too can be bypassed using techniques like return-to-libc and ROP-based exploits.

Bubble [37] is an effective technique to mitigate heap spraying wherein diversity is introduced at random locations in the heap. Inserting special interrupting values in strings at random locations in the heap at the time when a string is stored in memory breaks the attackers' assumption of memory homogeneity, thereby preventing heap spraying attack.

JITDefender [13] is one way of improving JIT compiler security. It hinders JIT spraying attacks by making all HLL code pages as non-executable. These pages are changed to executable by the compiler once the control enters in the generated code and are changed back on return to the compiler on runtime. This prevents an attacker from redirecting any other branch to dynamically generated code. INSeRT [22] add code randomization through a white box approach, requiring manual changes in the code generation phase of compiler.

Control Flow Integrity was first proposed by Abadi et al.[14]. This technique marks the valid targets of indirect

control transfers (i.e. function entry points and landing points for returns) with unique identifiers(IDs) and then inserts ID-checks before each indirect call or return instruction. They then identify the set of valid targets through a precise control flow graph construction and enforcing control flow only to this set for each indirect transfer instruction. This technique was improved by Zhang et al in [23].

DynamoRIO [24], Valgrind [25], PIN [27] and Strata [26] are binary rewriting tools. These tools are used for increasing security, profiling, instrumentation and debugging.

VII. Conclusion and Future Work

In this contribution, we presented `libmask`, a non-XOR based Constant Masking technique for protecting browser JIT engines against JIT spraying attacks that leverage constants. `libmask` is capable of masking constants of all sizes, while incurring a modest overhead over performance. `libmask` replaces constant in the machine code with randomized addresses. We also presented several optimizations for decreasing the performance overhead.

We integrated `libmask` with Google Chrome's V8 JavaScript Engine. We evaluated the performance and security impact of our technique by executing Octane Benchmarks and compared it with existing mechanism, i.e. XOR based constant blinding. We found that our approach masks all constants in the JITed code and the average performance overhead we obtained was 5.31%. We also found out, our approach not only performs better than XOR based constant blinding but also has better security impact than XOR based constant blinding.

As a part of our future work, we intend to further enhance the performance of `libmask` and explore the possibility of developing hybrid mechanisms presented in academia and industries to harden security against JIT-ROP attacks.

References

[1] F. Cohen. Operating system protection through program evolution. Computers and Security, 12(6):565–584, Oct. 1993.

[2] S. Forrest, A. Somayaji, and D. Ackley. Building diverse computer systems. In Proceedings of the 6th Workshop on Hot Topics in Operating Systems, HotOS '97, pages 67–72, 1997.

[3] C. Giuffrida, A. Kuijsten, and A. S. Tanenbaum. Enhanced operating system security through efficient and fine-grained address space randomization. In Proceedings of the 21st USENIX Security Symposium, pages 475–490, 2012.

[4] 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.

[5] Dion Blazakis. Interpreter Exploitation: Pointer Inference and JIT Spraying. http://www.semantiscope.com/research/BHDC2010/BHDC-2010-Paper.pdf

[6] T. Jackson, B. Salamat, A. Homescu, K. Manivannan, G. Wagner, A. Gal, S. Brunthaler, C. Wimmer, and M. Franz. Compiler-generated software diversity. In S. Jajodia, A. K. Ghosh, V. Swarup, C. Wang, and X. S. Wang, editors, Moving Target Defense, volume 54 of Advances in Information Security, pages 77–98. Springer New York, 2011.

[7] A. Homescu, S. Neisius, P. Larsen, S. Brunthaler, and M. Franz. Profile-guided automated software diversity. In Proceedings of the 2013 International Symposium on Code Generation and Optimization, CGO '13, 2013.

[8] A. Homescu, S. Brunthaler, P. Larsen, and M. Franz. librando: Transparent Code Randomization for Just-in-Time Compilers In ACM Conference on Computer and Communications Security, CCS' 13.

[9] J. Hiser, A. Nguyen-Tuong, M. Co, M. Hall, and J. W. Davidson. ILR: Where'd my gadgets go? In Proceedings of the 33rd IEEE Symposium on Security and Privacy, S&P '12, pages 571–585, 2012.

[10] V. Pappas, M. Polychronakis, and A. D. Keromytis. Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. In Proceedings of the 33rd IEEE Symposium on Security and Privacy, S&P '12, pages 601–615, 2012.

[11] 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 19th ACM Conference on Computer and Communications Security, CCS '12, pages 157–168, 2012.

[12] C. Song, C. Zhang, T. Wang, W. Lee, and D. Melski, Exploiting and protecting dynamic code generation, in Proceedings of the 2015 Network and Distributed System Security (NDSS) Symposium, February 2015.

[13] P. Chen, Y. Fang, B. Mao, and L. Xie. JITDefender: A defense against JIT spraying attacks. In Proceedings of the 26th IFIP TC 11 International Information Security Conference, SEC '11, pages 142–153, 2011.

[14] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, Control-flow integrity, in ACM Conference on Computer and Communications Security (CCS), 2005.

[15] Browser Statistics and Trends. http://www.w3schools.com/browsers/browsers_stats.asp

[16] Octane 2.0 JavaScript Benchmark. http://octane-benchmark.googlecode.com/svn/latest/index.html

[17] M. Athanasakis, E. Athanasopoulos, M. Polychronakis, G. Portokalidis, S. Ioannidis, The Devil is in the Constants: Bypassing Defenses in Browser JIT Engines in Network and Data Security Symposium, NDSS' 15.

[18] H. Shacham, M. Page, B. Pfaff, E. 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, CCS '04, pages 298–307, 2004.

[19] J. Salwan. ROPgadget tool, 2012. URL http://shell-storm.org/project/ROPgadget/.

[20] V8 Constant Blinding `SafePush` https://github.com/v8/v8/blob/master/src/ia32/macro-assembler-ia32.cc#L607

[21] V8 Constant Blinding `SafeMove` https://github.com/v8/v8/blob/master/src/ia32/macro-assembler-ia32.cc#L597

[22] T. Wei, T. Wang, L. Duan, and J. Luo. INSeRT: Protect dynamic code generation against spraying. In Proceedings of the 2011 International Conference on Information Science and Technology, ICIST '11, pages 323–328, 2011

[23] Practical control flow integrity and randomization for binary executables. In IEEE Symposium on Security and Privacy, S&P '13, pages 559 - 573, 2013

[24] D. Bruening, T. Garnett, and S. Amarasinghe. An infrastructure for adaptive dynamic optimization. In Proceedings of the 2003 International Symposium on Code Generation and Optimisation, CGO '03, 2003.

[25] N. Nethercote and J. Seward. Valgrind: A program supervision framework. Electronic Notes in Theoretical Computer Science, 2003.

[26] K. Scott, N. Kumar, S. Velusamy, B. R. Childers, J. W. Davidson, and M. L. Soffa. Retargetable and reconfigurable software dynamic translation. In Proceedings of the 2003 International Symposium on Code Generation and Optimisation, CGO '03, 2003.

[27] 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. In Proceedings of the ACM PLDI '05, pages 190–200, 2005.

[28] Microsoft Visual Studio 2005, Image has safe exception handlers, http://msdn.microsoft.com/en-us/library/9a89h429%28v=vs.80%29.aspx.

[29] A. Jangda, M. Mishra, and B. De Sutter. adaptive: Adaptive Just-In-Time Code Diversification. In Proceedings of MTD, ACM CCS'15, Denver CO, USA, Oct. 2015.

[30] https://developers.google.com/v8/?hl=en

[31] URL https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey

[32] URL https://github.com/Microsoft/ChakraCore

[33] URL http://trac.webkit.org/wiki/JavaScriptCore

[34] URL https://developer.mozilla.org/en-US/docs/Mozilla/Projects/Rhino

[35] Aleph One. Smashing the Stack for Fun and Profit. Phrack 49. URL http://insecure.org/stf/smashstack.html

[36] skypher.com. Heap spraying (2007), http://skypher.com/wiki/index.php

[37] F. Gadaleta , Y. Younan , W. Joosen. BuBBle: A Javascript Engine Level Countermeasure against Heap-spraying Attacks. Proc. 2nd Int'l Symp. Eng. Secure Software and Systems, 2010.

[38] V8 Runtime Functions https://github.com/v8/v8/blob/master/src/runtime/runtime-scopes.cc