Look Ma, No Constants: Practical Constant Blinding in GraalVM

Felix Berlakovich
Bundeswehr University Munich
Germany
felix.berlakovich@unibw.de

Matthias Neugschwandtner
Oracle Labs, Austria
Austria
matthias.neugschwandtner@oracle.com

Gergö Barany
Oracle Labs, Austria
Austria
gergo.barany@oracle.com

ABSTRACT

With the advent of JIT compilers, code-injection attacks have seen a revival in the form of JIT spraying. JIT spraying enables an attacker to inject gadgets into executable memory, effectively sidestepping W⊕X and ASLR.

In response to JIT spraying, constant blinding has emerged as a conceptually straightforward and performance friendly defense. Unfortunately, increasingly sophisticated attacks have pinpointed the shortcomings of existing constant blinding implementations.

In this paper we present our constant blinding implementation in the GraalVM compiler, enabling constant blinding across a wide range of languages. Our implementation takes insights from the last decade of research on the security of constant blinding into account. We discuss important design decisions and trade-offs as well as the practical implementation issues encountered when implementing constant blinding for GraalVM. We evaluate the performance impact of our implementation with different configurations and demonstrate its effectiveness by fuzzing for unblinded constants.

CCS CONCEPTS

• **Security and privacy → Systems security**.

KEYWORDS

constant blinding, JIT spraying, JIT compilation, language runtimes

ACM Reference Format:
Felix Berlakovich, Matthias Neugschwandtner, and Gergö Barany. 2022. Look Ma, No Constants: Practical Constant Blinding in GraalVM. In *15th European Workshop on Systems Security (EUROSEC '22), April 5–8, 2022, RENNES, France.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3517208.3523751

1 INTRODUCTION

Over the last decade, just-in-time (JIT) compilers have continuously gained in popularity by boosting the performance of language implementations into unprecedented heights [5]. JIT compilers improve performance by compiling the input program to native machine code just in time, that is, at runtime. This runtime code generation, however, has resurrected a type of attack that was already believed to be a relic of the past: code injection. In the context of JIT compilers, code injection is called JIT spraying and allows an attacker to control not only the program inputs, but also, to a certain extent, the generated machine code. At the core of JIT spraying is the attacker's ability to predict the machine code resulting from carefully crafted input programs. The JIT spraying attack was initially demonstrated on Adobe's ActionScript JIT compiler but has since been ported to different languages VMs, such as the Linux eBPF VM, Webkit JSC, or Spidermonkey [6, 19, 25, 26].

One of the most popular language VMs is the Java HotSpot VM. With GraalVM, a single compiler allows Java and a wide range of other languages to target the HotSpot VM as well as native libraries and executables that can be embedded in a variety of scenarios (see Section 2.2). GraalVM's polyglot nature and its ability to be embedded into other software makes it an attractive choice for embedders. To strengthen GraalVM's resilience against untrusted code execution, we add constant blinding to the GraalVM compiler.

Constant blinding is a battle-proven defense against JIT spraying, which invalidates attacker predictions by adding randomness to the compilation process. Although constant blinding is conceptually straightforward, a wealth of attacks demonstrates that the devil is in the details [4, 19–22, 28].

In this paper we present our implementation of constant blinding in the GraalVM compiler, which is part of the GraalVM Enterprise Edition starting with version 21.3.0. We explain the general design decisions and pitfalls and show how our implementation relates to known attacks against constant blinding. In addition, we show how our implementation deals with HotSpot's peculiarities and how we integrate constant blinding into the GraalVM compiler's optimization pipeline.

In summary, we make the following contributions:

- We implement constant blinding in a polyglot runtime, bringing it to languages that were lacking JIT spraying mitigations so far.
- We discuss important design decisions, highlight issues faced in practice and explain how we tackled them in our implementation.
- Our implementation of constant blinding can be tuned based on a risk/performance trade-off by allowing embedders to specify the minimum size of constants that will be blinded.
- We evaluate both the performance overhead of our implementation across different configurations as well as its effectiveness.

2 BACKGROUND

2.1 JIT spraying

The widespread adoption of W⊕X and ASLR has considerably raised the bar for attackers to mount a successful attack. Unfortunately, in the context of JIT compilation the effectiveness of both defenses is weakened significantly. JIT compilers allow users to provide source

EUROSEC '22, April 5–8, 2022, RENNES, France

ACM ISBN 978-1-4503-9255-6/22/04…$15.00
https://doi.org/10.1145/3517208.3523751

code and, given the source code is valid, compile it to machine code. From an attacker's perspective, JIT compilers compile attacker-controlled inputs to predictable bytes in executable memory.

In an attack called *JIT spraying* an attacker leverages the predictable compilation by feeding malicious input programs into the JIT compiler, thereby forcing it to emit code containing ROP gadgets [6]. JIT spraying is problematic for two reasons. First, with JIT spraying an attacker is no longer limited to gadgets already present in the attacked process. JIT spraying enables an attacker to inject new gadgets into executable memory, thus bypassing W⊕X, which only protects statically compiled parts of the application. Second, JIT spraying allows an attacker to generate a large number of gadget copies. As each additional gadget copy increases the odds of hitting a gadget by chance, JIT spraying effectively weakens the security guarantees afforded by ASLR. With an abundance of gadget copies available an attacker no longer needs to know the exact gadget location, but can guess with a high chance of success.

Since its initial demonstration in 2010 [6], the JIT spraying attack has been the subject of active research. Different kinds of defenses [1, 8, 13, 15, 16, 22] stand against increasingly resilient attack variants [4, 19–21]. Gawlik and Holz provide a more detailed overview of attacks on JIT compilers in general and JIT spraying in particular [14].

2.2 GraalVM

GraalVM is a suite of polyglot runtime environment technologies from Oracle Labs, built around the GraalVM compiler. The GraalVM compiler can target the HotSpot Java VM, replacing its standard JIT compiler. Alternatively, GraalVM can build *native images*: A native image contains a heap snapshot of an application after startup as well as all reachable code compiled ahead-of-time using the GraalVM compiler [29]. The heap snapshot and the compiled code are then bundled as a binary executable or shared library together with a component called SubstrateVM. SubstrateVM implements the necessary runtime services (e.g., garbage collector) otherwise provided by a Java VM.

The GraalVM compiler directly compiles JVM bytecode for Java and other JVM languages like Scala. Further polyglot capabilities are provided using Truffle, a framework for implementing languages. Truffle performs *partial evaluation* on language implementations, transforming each language's code into a form that can be directly compiled to high-performance machine code using the GraalVM compiler [31]. GraalVM comes with a number of Truffle language implementations, including Javascript, Python, Ruby, and LLVM IR. Native images of either Java code or Truffle language engines can be run stand-alone or embedded in regular native code applications as shared libraries.

GraalVM is available as a closed source enterprise edition as well as an open source community edition[1].

3 THREAT MODEL

Our constant blinding implementation for the GraalVM compiler builds on the following assumptions and threat model:

- The adversary has access to a memory corruption vulnerability that enables control-flow hijacking.
- The adversary can freely choose the source code to execute on GraalVM.
- The adversary knows the system configuration of the target system. Specifically, the adversary knows the exact version and configuration of GraalVM as well as the target system's architecture.
- The system is protected against conventional code injection with W⊕X or DEP.

In this paper we focus on the danger emanating from attacks that leverage JIT spraying with constants to *inject* gadgets into executable memory. We do not consider gadgets found in instructions not related to constants. Similarly, attacks that chain gadgets found outside of JIT compiled code (e.g., conventional code-reuse attacks) are out of scope.

4 GRAALVM CONSTANT BLINDING

In Section 2.1 we saw that attackers can abuse the predictable compilation of programs fed into the JIT compiler to control bytes in executable memory. Constants in the input program are a particularly attractive target for such an attack, since JIT compilers often include them *verbatim* in the machine code. Constant blinding aims to invalidate an attacker's predictions by introducing randomness into the compilation process [18]. Specifically, constant blinding encrypts constants with a random key at compile time and decrypts them at runtime at each occurrence. Only the encrypted version of the constant appears verbatim in the machine code. Absent knowledge of the random key, the attacker cannot predict the encrypted constant value and, therefore, can no longer predict the resulting bytes in executable memory.

While the principle behind constant blinding is straightforward, a practical implementation faces interesting design decisions and pitfalls. In the following sections we detail how we implement constant blinding for the GraalVM compiler.

4.1 How to blind

Our constant blinding implementation consists of a phase in the GraalVM compiler's compilation pipeline. During compilation, the compiler represents the input program as a graph called HIR graph [11]. Within the HIR graph, constant values in the input program are represented by Constant nodes. Our phase replaces Constant nodes in the graph with BlindedConstant nodes. During the later code generation phase, a BlindedConstant node in the graph causes the compiler to (a) generate an encrypted constant at compile time and (b) generate machine code to decrypt the constant at runtime.

In principle, any encryption function $f(c)$ for which $f^{-1}(c)$ exists can be used to blind and unblind constants, respectively. Since the decryption happens at runtime, however, the choice of $f^{-1}(c)$ directly influences the performance impact. Like other constant blinding implementations, we chose XOR as a performance friendly encryption and decryption function. We implement the runtime decryption using the CPU's xor instruction, which typically has a low cycle count and latency. In total, two instructions are required to decrypt an encrypted constant. For example, if the constant has been encrypted with the random key 0xABCD and is located in the register rdi, the following code decrypts the constant:

```
mov rax, 0xABCD  // load the key
```

[1] https://github.com/oracle/graal/

```
public static void attack() {
  final int evil = 0xc358; ①
  array[0xc358 - 16] = 0xa; ②
  fields.f_c358 = 0xb; ③
}
class Fields {
  byte f_000c, f_000d, f_000e, ..., f_c358;
}
public static Fields fields = new Fields();
```

```
mov rax, 0xc358 ①
...
; pointer to array in ebx
mov BYTE PTR [ebx*8+0xc358], 0xa ②
...
; pointer to fields in ebx
mov BYTE PTR [ebx*8+0xc358], 0xb ③
```

Figure 1: The compilation of a Java method `attack` containing attacker-controlled constant values (left) to machine code (right). The constants are compiled to immediate values or to displacements for a array/field access (see Section 4.2.1 for details).

```
xor rdi, rax      // decrypt the constant
```

Despite the efficiency of XOR, decrypting a large number of constants impacts performance. To reduce the performance impact, prior constant blinding implementations blinded only constants with three bytes or more [4]. The assumption behind this size restriction is that 1- or 2-byte constants cannot encode useful code-reuse gadgets. Athanasakis et al. show, however, that this assumption does not hold in practice [4]. Therefore, we support blinding of all constant sizes, but allow the user to configure the minimum size of blinded constants. We discuss the performance impact of different blinded constant sizes in Section 5.1.

Another way to decrease the performance impact of constant blinding is to blind only a random subset of all constants. For example, Webkit's JSC blinds only one out of 64 constants on average. Unfortunately, Lian et al. show that leaving constants unblinded might open the door to a successful attack [19].

4.2 What to blind

Most of the bytes emitted by the JIT compiler are fixed by the input language semantics and, thus, not useful to an attacker. For example, an add operation in the source program is typically compiled to an `add` machine instruction. For a byte sequence to be useful during an attack, the attacker must be able to control its value. As an example, consider a constant integer variable in the input program. Such a constant variable is typically compiled to an immediate value inlined into the machine code. Since an attacker controls constants in the input program, she also controls the corresponding immediate byte sequences located in executable memory.

Machine code immediates produced by constant values in the input program are the most prominent example of attacker-controlled byte sequences. However, prior work has demonstrated the importance of considering other instances of attacker-controlled byte sequences as well [21, 28]. Our constant blinding implementation considers the following instances of attacker-controlled byte sequences in the GraalVM compiler:

- constants compiled to inline immediate values in machine code;
- constants embedded into executable memory as data;
- composite constants.

In the following subsections we explain each of these cases in more detail. Our running example is an attacker trying to embed the byte sequence `58 c3`, which encodes the x86-64 instructions for the ROP gadget `pop rax; ret`.

4.2.1 Immediate values. JIT compilers often compile constant values in the input program to immediates inlined in the machine code. As an example consider the Java method `attack` and its compiled version in Figure 1. The value of the constant `evil` is embedded verbatim into the machine code (see ①). Due to x86-64's little-endian byte order, the embedded value $0xc358$ will be encoded as the hexadecimal sequence `58 c3 00 00` in order of increasing memory addresses.

Another example is the constant array index expression (see ②). Note that the index expression uses a value 16 bytes smaller than the desired gadget to account for the array header offset that is automatically added by the compiler.

In the third example the desired constant does not appear verbatim in the input program. Instead, the attacker leverages a field access to a specially crafted class (see ③). The machine code of such an access contains an immediate value that encodes the offset of the field within the class. Since the attacker can supply an arbitrary class, she can control the exact offset used in the machine code. The byte-sized fields are numbered from $0x000c$ to account for the 12-byte object header.

Our constant blinding implementation handles these cases uniformly by blinding all attacker controlled constants, including address offsets, that produce machine code immediates.

4.2.2 Data embedded in code pages. Immediates in the machine code are generally limited in size. For example, with the exception of `movabs`, all instructions on the x86-64 platform support only 32 bit immediates. Immediates on ARM are even limited to 12 bits. As a result, code operating on constant values larger than the supported immediate size must load these constants from memory. For example, instead of using immediates, the GraalVM compiler compiles accesses to floating point values as PC-relative memory accesses. To keep the PC-relative offsets short and the resulting instructions compact, the target VM may choose to embed the constant data into the code page.

Another type of data that can be embedded into code pages are *frame states*. Frame states are part of GraalVM's speculative optimization infrastructure. The compiler can compile code speculatively under assumptions like certain exceptions never being thrown or certain parts of the code never being reached. If such an assumption ever becomes invalid, in a process called *deoptimization*, the compiled code transfers execution back to an interpreter. Deoptimized methods can be recompiled later without the failed assumptions. Frame states encode the information necessary to enable the transfer of control, namely the program point and the values of local variables at that point. The content of frame states is, thus, partly under attacker control and can contain attacker provided constants. While SubstrateVM stores frame states in a read-only data page, HotSpot embeds them into executable code pages next to the machine code.

Unfortunately, embedding data in a code page exposes the data to JIT spraying. Prior work has demonstrated the feasibility of abusing constants embedded into code pages [4]. Strictly separating code from data would require a significant change to HotSpot's code base and cannot be done within the GraalVM compiler alone. For instance, the separation of code and metadata like frame states is a long standing issue [17]. To mitigate JIT spraying, our implementation blinds constant data embedded into code pages and, when compiling for HotSpot, also constants occurring in frame states.

4.2.3 Composite constants. Certain instructions can contain more than one embedded constant, and these constants may be adjacent in the instruction encoding. An attacker can exploit this to fabricate constants larger than the constant blinding size limit [28]. Consider the example from Shinagawa et al. of an operation writing a constant byte into a byte array at a constant offset:

```
bytes[0x58 - 16] = (byte) 0xc3;
```

This can compile directly to the following instruction:

```
mov BYTE PTR [rsi+0x58], 0xc3
```

The straightforward encoding of this instruction is `c6 46 58 c3`, containing our sample ROP gadget. This gives an attacker control over a 2-byte constant concatenated from 1-byte constants, so blinding constants ≥ 2 bytes is an insufficient defense. In general this situation is difficult to detect in the HIR graph because the constants are inputs to different nodes, and we would need to predict how they will be combined to a single instruction. While implementing the detection of simple patterns such as this one is possible, a hardcoded pattern detection is tedious and error prone.

The GraalVM compiler's embedded assembler has a feature for always emitting address displacements as 4-byte constants, even if they would fit into a single byte. We use this as a partial mitigation against the 1-byte constant attack: If constant blinding for constants ≥ 2 bytes is enabled, we force the assembler to always emit 4-byte displacements. The above instruction is then encoded as `c6 86 58 00 00 00 c3`, which does not contain the same gadget. This sequence does still contain `00 00 c3`, which encodes `add BYTE PTR [rax], al; ret`. These instructions might still be useful to an attacker, but with her choices severely restricted.

This approach also tears apart Shinagawa et al.'s 3-byte examples, with the `ret` separated from the other instructions.

Padding displacements slightly increases the code size and leads to a small performance degradation (see Section 5.1). Alternatively, our implementation allows the user to trade performance overhead for additional security by blinding even 1-byte constants. In the example above, blinding 1-byte constants renders all the bytes in the gadget unpredictable for the attacker.

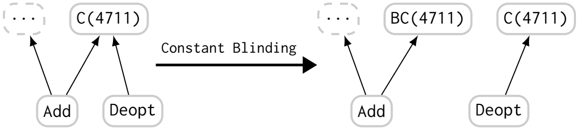

Figure 2: Blinding of a deduplicated Constant node C based on its usages. After blinding, the Add node uses a BlindedConstant node BC, whereas the Deoptimize node Deopt uses the original Constant node. See Section 4.3.1 for details.

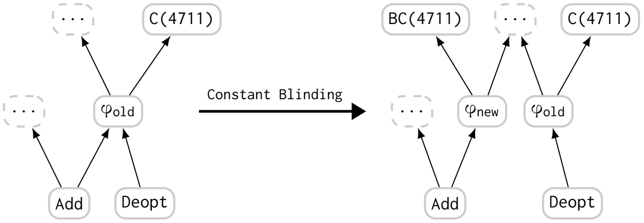

Figure 3: Blinding of a Constant node with a Phi node usage. After blinding the Phi node, each Phi node usage transitively uses the correct type of constant node. See Section 4.3.2 for details.

4.3 Exceptions from blinding

The Constant nodes in the HIR graph can be categorized into two groups. First, constants that can be controlled directly or indirectly by an attacker. We gave examples of such constants in Section 4.2.1, Section 4.2.2, and Section 4.2.3. Second, constants with a fixed meaning and value range inserted by the compiler. For example, the GraalVM compiler represents deoptimization points with Deoptimize nodes in the graph. These Deoptimize nodes receive a Constant node as input that encodes the deoptimization reason. As the value of these constants is not under attacker control, blinding is irrelevant.

Our compiler phase decides whether a Constant node needs blinding based on its use. If blinding is necessary, the phase replaces the Constant node with a BlindedConstant node. For example, the phase blinds Constant nodes used by an Add node, since the constants could result from an attacker-controlled addition in the input program. A Constant node with a Deoptimize node usage on the other hand is not blinded.

4.3.1 Constant Deduplication. The GraalVM compiler deduplicates equal nodes in the HIR graph. Deduplication not only keeps the graph compact, but also implements global value numbering [7]. Consider the two nodes in Figure 2 as an example. Deduplication leads to a single Constant node representing the inputs of both the Add node and the Deoptimize node. Since only the Add node should use a blinded constant, the constant blinding phase needs to deduplicate the Constant node. To deduplicate the Constant node, the phase first introduces a new BlindedConstant node representing the blinded constant value. Next, the phase updates all usages that should use a blinded constant to use the BlindedConstant node instead.

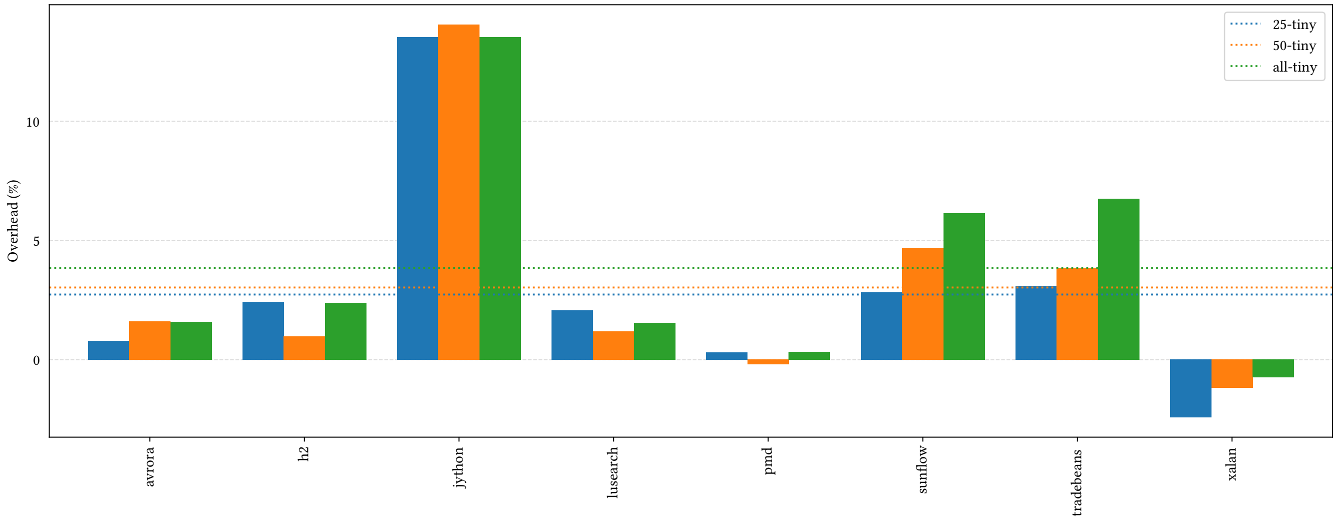

Figure 4: Overhead of three constant blinding sizes on execution time. The error bars were estimated with Fieller's theorem [12] with a confidence interval of 95%. The dotted lines represent the geometric mean with negative values excluded.

4.3.2 Phi Node Splitting. The HIR graph represents the program in SSA form and uses Phi nodes to select between multiple inputs at control-flow merges [10, 27]. Unfortunately, Phi nodes complicate the deduplication process. As an example, consider the graph in Figure 3. In the example, a Phi node receives, among other nodes, a Constant node as input. The Phi node's usages transitively use the Constant node through the Phi node. However, only *some* of the Phi node's usages should use a *blinded constant.* In such a case the constant blinding phase must deduplicate not only the Constant node, but also the Phi node. Phi node deduplication happens similarly to the deduplication of Constant nodes. First, the phase introduces a new Phi node with the same inputs as the old one. Next, the phase updates all the usages of the old Phi node that should use a blinded constant to use the new Phi node instead. As Phi nodes can act as inputs for other Phi nodes, the phase repeats the deduplication process recursively in a depth-first manner. Once all Phi nodes connected directly or indirectly to a Constant node are processed, the following invariant holds: *Each Phi node is connected (directly or transitively) to either a Constant node or a BlindedConstant node, but not both.*

Cycles of Phi nodes in the graph are resolved using worklist iteration, keeping a mapping from old Phi nodes to already deduplicated ones to ensure termination.

4.3.3 Compiler-generated constants. Most constants in the HIR graph are derived directly or via optimizations from the input program. These can be controlled by an attacker who controls the input code. Some other constants are independent of the input program and are introduced by the compiler based on information from the target VM. In particular, the code generated for tracking garbage collection metadata can include addresses of VM-allocated data areas as constants.

The GraalVM compiler's compilation pipeline is divided into high, mid, and low tiers, successively refining more abstract high-level operations to concrete machine-specific instructions. Abstract memory barrier operations for tracking garbage collection data are expanded to memory accesses in the low tier, adding new VM-specific constants to the graph. We consider constants that are present in the graph before the low tier to be potentially attacker controlled and subject to blinding. Constants only introduced during the low tier are exempt from blinding because we consider them not attacker controllable.

5 EVALUATION

5.1 Execution time overhead

We used version 0.11.0 of the standard Renaissance JVM benchmark suite [24] to evaluate the overhead of our constant blinding implementation on application execution time. The GraalVM distribution comes with benchmark tooling that is aware of this benchmark suite, runs each benchmark appropriately, and summarizes results.

Each benchmark is run for a number of iterations to account for virtual machine warmup and the benchmark's variability. We use the default numbers of iterations specified by the Renaissance benchmark and calculate the average of the last 40% of runs, but at least 6 and at most 20. All prior iterations serve as warmup. The numbers range from 4 warmup and 6 measurement iterations for the reactors benchmark to 70 + 20 iterations for finagle-chirper.

We used a development snapshot of GraalVM 22.0 Enterprise Edition for JDK11. The benchmarks were run on an otherwise unloaded multi-node Linux server, pinned to a single 18-core 2.3 GHz Intel Xeon E5-2699 v3 node.

We ran a total of four configurations: A baseline configuration with constant blinding disabled (as is the default on GraalVM), and one configuration each blinding constants ≥ 1, 2, and 4 bytes respectively. Figure 4 summarizes our results, showing execution times normalized to the baseline, both for individual benchmarks and as the geometric mean over all of them. Despite running the recommended numbers of repetitions of each benchmark and repeating the experiment, four benchmarks on the right (gauss-mix, page-rank, scala-doku, and scrabble) had repeatedly noisy results. For that reason we give our results with and without these noisy benchmarks.

Blinding constants ≥ 4 bytes had a geometric mean overhead of 2.83 % (maximum: 7.96 %), for ≥ 2 bytes we observed a mean overhead of 4.30 % (maximum: 13.35 %) and for ≥ 1 byte we observed a mean overhead of 16.15 % (maximum: 34.08 %). With the noisy benchmarks excluded, we observe a slightly reduced mean overhead of 2.83 %, 4.21 % and 15.69 % respectively. These numbers include the mitigation against composite constants (Section 4.2.3).

The reason for the much larger impact of 1 byte constants is that these are especially frequent: Looking at the dotty benchmark as a representative example, we find that 64970 out of 87895 constants (74 %) are bytes. Of these byte-sized constants, 41 % have at least one use as an address offset, typically as object field offsets or array indices.

5.2 Effectiveness

To evaluate the effectiveness of our constant blinding implementation, we adopted the idea of Dachshund [22]. Dachshund is a fuzzer that searches for unblinded constants in JavaScript language VMs. Since the original Dachshund implementation was not available, we implemented a corresponding fuzzer with the ability to search for unblinded constants ourselves. Specifically, our fuzzer generates random Java programs containing random constants and compiles the programs with the GraalVM compiler. After the compilation, the fuzzer searches for the constants in the GraalVM compiler's output. Note that the fuzzer scans the generated machine code as well as data embedded into code pages. When a constant C occurs as an array index, the fuzzer additionally searches for $C + 16$ to account for the array header offset (see Section 4.2.1).

We found that fuzzing for unblinded constants works reliably for constants with a minimum size of three bytes or more. Without constant blinding, the fuzzer found a large proportion of the random constants in the input program occurring verbatim in the generated machine code. Conversely, with constant blinding enabled, the fuzzer found no unblinded constants of the given minimum size in 20,000 generated programs.

For random 2-byte constants the corresponding bytes occur with a high probability at random locations anywhere in the instruction stream. Such occurrences are not useful for our purpose because we focus explicitly on gadgets resulting from attacker controllable embedded constants (see Section 3). To still evaluate the effectiveness of our implementation we adapted the fuzzer for the case of 2-byte constants. When testing these small constants, instead of generating constants with random bytes, the fuzzer uses *fixed byte sequences* that resemble invalid x86-64 opcodes[2]. As a result, the byte sequences no longer appear randomly in the instruction stream. The only places where the fixed byte sequences can occur are in control-flow offsets, memory addresses, and actually unblinded constants. We do not currently blind control-flow offsets (see Section 6), and occurrences in memory addresses are not predictable thanks to ASLR. When excluding control-flow offsets and memory addresses from the results, our fuzzer effectively uncovers unblinded 2-byte constants if constant blinding is disabled. In contrast, when blinding constants with a minimum size of two bytes, we found no unblinded constants in 20,000 generated programs.

Searching only for constants that appear verbatim in the *input program* would not account for composite constants (see Section 4.2.3). For example, assume that the fuzzer generated the array access `bytes[0x06] = (byte)0x1e`. Recall from Section 4.2.3 that such an input program can lead to the bytes $0x06 + 16 = 0x16$ and $0x1e$ being adjacent in the machine code. To find adjacent constant bytes, our fuzzer also searches for concatenations of 1- and 2-byte constants. In the example the fuzzer would search for $0x161e$ and $0x1e16$, effectively uncovering the adjacent 1-byte constants. We found that blinding constants with a minimum size of one byte left no unblinded constants in 20,000 generated programs.

6 DISCUSSION & RELATED WORK

Our performance evaluation for blinding all constant sizes contradicts an overhead estimation of up to 80 % as reported in the literature [4]. We speculate that our better performance has two main reasons. First, Athanasakis et al. consider *all* instructions with immediate values, whereas our implementation focuses on immediate values actually controllable by an attacker (see Section 4.3.3). Second, Athanasakis et al. estimate the overhead by summing the required cycles of all instructions involved. Since modern CPUs are both superscalar and pipelined, such an estimate is overly conservative.

Maisuradze et al. report an overhead of 21% on top of the underlying constant blinding implementation [22]. Their approach is not directly comparable to our implementation, however. Instead of blinding constants of all sizes, Dachshund hardens the underlying constant blinding implementation for ≥ 4 bytes by rewriting JavaScript code with a proxy.

Lian et al. provide a constant blinding implementation for SpiderMonkey and report a remarkably low overhead of 1.39% [20]. Their implementation blinds constants in SpiderMonkey's IR. Based on our understanding of the IR, the lower overhead could result from the high abstraction level of certain IR instructions (e.g., `JSOP_GETPROP`). Constants used as part of the lowering of these high level instructions are not blinded.

Librando provides code randomization, including constant blinding, without modifying the JIT compiler [15]. Since the different randomization techniques were not measured in isolation, no direct comparison is possible. The numbers suggest, however, that librando's blackbox implementation of constant blinding leads to a higher performance overhead. With RIM, Wu et al. provide a defense against JIT spraying that splits 4-byte immediate values into two 2-byte sequences [30]. As discussed previously, however, 2-byte constants can still be used in an attack. JITSafe provides a JIT spraying mitigation for the Tamarin Flash Engine [8]. JITSafe prevents an abuse of constants by loading the constant values from the heap instead of inlining them as machine-code immediates.

Our implementation does not currently blind constant offsets used in control-flow instructions (e.g., `jmps` and `calls`). Maisuradze

[2] We use the values `0x07`, `0x27`, `0x1e` and `0x1f`.

et al. show that an attacker can exploit such constants by carefully crafting code with predictable offsets [21]. The authors propose to blind control-flow offsets similarly to data constants.

Constant blinding generally does not protect against gadgets arising from the interpretation of bytes of adjacent instructions. To deal with these gadgets, software diversity and CFI have emerged as promising ideas. Software diversity builds on the observation that abusing gadgets critically hinges on the exact knowledge of the generated code layout. Attackers know the code layout either because the code generation is predictable or because the attacker leaks the code layout through a memory disclosure vulnerability. Leakage-resilient software diversity, i.e., a combination of code layout randomization with a protection against information disclosure, protects against both of these attack vectors. With Readactor, Crane et al. demonstrate the applicability to JIT compilers [9]. Leakage-resilient diversity could also prove effective against the above-mentioned control-flow offset attack. CFI on the other hand confines control-flow transfers to an approximation of the control-flow graph determined through static analysis [2, 3]. RockJIT, for example, extends the idea of control-flow diversity to JIT compiled code [1]. We leave the implementation of a more general defense against ROP gadgets for future work.

Another option to reduce the attack surface of language VMs is to shift the focus from JIT compilation towards interpretation. Microsoft currently pursues this avenue with an experimental feature in the Edge browser called "Super Duper Secure Mode" [23]. Enabling this feature limits Edge's JavaScript engine to interpretation by disabling the JIT compiler, effectively trading performance for increased security.

7 CONCLUSION

Owing to their widespread adoption, JIT compilers have become a prime attack target and the JIT spraying attack in particular has gained sophistication. Constant blinding, although not a silver bullet, is a valuable tool when defending against JIT spraying attacks.

We have presented our implementation of constant blinding for GraalVM Enterprise Edition, taking into account the collective insights of prior work over the past decade. Given GraalVM's polyglot nature, our implementation brings constant blinding to a multitude of JIT compiled languages that lacked constant blinding so far. Our performance evaluation shows that our implementation imposes a geometric mean overhead of about 3 %, 4 % and 16 % respectively, depending on the minimum size of blinded constants. Our measured worst case overheads are much less prohibitive than prior pessimistic overhead estimations of up to 80 %.

REFERENCES

[1] 2014. RockJIT: Securing Just-In-Time Compilation Using Modular Control-Flow Integrity. In *ACM SIGSAC Conference on Computer and Communications Security.*

[2] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. 2005. A Theory of Secure Control Flow. In *International Conference on Formal Methods and Software Engineering.*

[3] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. 2009. Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information and System Security* 13, 1 (Oct. 2009), 1–40.

[4] Michalis Athanasakis, Elias Athanasopoulos, Michalis Polychronakis, Georgios Portokalidis, and Sotiris Ioannidis. 2015. The Devil is in the Constants: Bypassing Defenses in Browser JIT Engines. In *Network and Distributed System Security Symposium.*

[5] John Aycock. 2003. A brief history of just-in-time. *Comput. Surveys* 35, 2 (jun 2003), 97–113.

[6] Dionysus Blazakis. 2010. Interpreter Exploitation.. In *USENIX Workshop on Offensive Technologies.*

[7] Preston Briggs, Keith D Cooper, and L. Taylor Simpson. 1997. Value Numbering. *Software—Practice & Experience* 27, 6 (1997), 701–724.

[8] Ping Chen, Rui Wu, and Bing Mao. 2013. JITSafe: a framework against Just-in-time spraying attacks. *IET Information Security* 7, 4 (dec 2013), 283–292.

[9] Stephen Crane, Christopher Liebchen, Andrei Homescu, Lucas Davi, Per Larsen, Ahmad-Reza Sadeghi, Stefan Brunthaler, and Michael Franz. 2015. Readactor: Practical Code Randomization Resilient to Memory Disclosure. In *IEEE Symposium on Security and Privacy.*

[10] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. 1991. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems* 13, 4 (oct 1991), 451–490.

[11] Gilles Duboscq, Thomas Würthinger, Lukas Stadler, Christian Wimmer, Doug Simon, and Hanspeter Mössenböck. 2013. An intermediate representation for speculative optimizations in a dynamic compiler. In *ACM workshop on Virtual machines and intermediate languages.* New York, New York, USA.

[12] E C Fieller. 1954. Some Problems in Interval Estimation. *Journal of the Royal Statistical Society. Series B (Methodological)* 16, 2 (1954), 175–185.

[13] Tommaso Frassetto, David Gens, Christopher Liebchen, and Ahmad-Reza Sadeghi. 2017. JITGuard: Hardening Just-in-time Compilers with SGX. In *ACM SIGSAC Conference on Computer and Communications Security.*

[14] Robert Gawlik and Thorsten Holz. 2018. SoK: Make JIT-Spray Great Again. In *USENIX Workshop on Offensive Technologies.*

[15] Andrei Homescu, Stefan Brunthaler, Per Larsen, and Michael Franz. 2013. Librando: transparent code randomization for just-in-time compilers. In *ACM SIGSAC Conference on Computer & Communications security.*

[16] Martin Jauernig, Matthias Neugschwandtner, Christian Platzer, and Paolo Milani Comparetti. 2014. Lobotomy: An Architecture for JIT Spraying Mitigation. In *International Conference on Availability, Reliability and Security.*

[17] Vladimir Kozlov. [n. d.]. *JDK-7072317.* https://bugs.openjdk.java.net/browse/JDK-7072317

[18] Per Larsen, Andrei Homescu, Stefan Brunthaler, and Michael Franz. 2014. SoK: Automated Software Diversity. In *IEEE Symposium on Security and Privacy.*

[19] Wilson Lian, Hovav Shacham, and Stefan Savage. 2015. Too LeJIT to Quit: Extending JIT Spraying to ARM. In *Network and Distributed System Security Symposium.*

[20] Wilson Lian, Hovav Shacham, and Stefan Savage. 2017. A Call to ARMs: Understanding the Costs and Benefits of JIT Spraying Mitigations. In *Network and Distributed System Security Symposium.*

[21] Giorgi Maisuradze, Michael Backes, and Christian Rossow. 2016. What Cannot be Read, Cannot be Leveraged? Revisiting Assumptions of JIT-ROP Defenses. In *USENIX Security Symposium.*

[22] Giorgi Maisuradze, Michael Backes, and Christian Rossow. 2017. Dachshund: Digging for and Securing (Non-)Blinded Constants in JIT Code. In *Network and Distributed System Security Symposium.*

[23] Microsoft. [n. d.]. *Super Duper Secure Mode.* https://microsoftedge.github.io/edgevr/posts/Super-Duper-Secure-Mode/

[24] Aleksandar Prokopec, Andrea Rosà, David Leopoldseder, Gilles Duboscq, Petr Tůma, Martin Studener, Lubomír Bulej, Yudi Zheng, Alex Villazón, Doug Simon, Thomas Würthinger, and Walter Binder. 2019. Renaissance: Benchmarking Suite for Parallel Applications on the JVM. In *ACM SIGPLAN Conference on Programming Language Design and Implementation.*

[25] Elena Reshetova, Filippo Bonazzi, and N Asokan. 2016. Randomization can't stop BPF JIT spray. In *Black Hat USA.*

[26] Chris Rohlf and Yan Ivnitskiy. 2011. Attacking clientside JIT compilers. In *Black Hat USA.*

[27] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. 1988. Global value numbers and redundant computations. In *ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '88.*

[28] Takahiro Shinagawa, Yuki Suzuki, Tomoyuki Nakayama, and Masanori Misono. 2019. The 1-Byte Constant Attack against JIT Compilers. (2019).

[29] Christian Wimmer, Codrut Stancu, Peter Hofer, Vojin Jovanovic, Paul Wögerer, Peter B. Kessler, Oleg Pliss, and Thomas Würthinger. 2019. Initialize Once, Start Fast: Application Initialization at Build Time. *Proceedings of the ACM on Programming Languages* 3, OOPSLA, Article 184 (oct 2019), 29 pages.

[30] Rui Wu, Ping Chen, Bing Mao, and Li Xie. 2012. RIM: A method to defend from JIT spraying attack. *Proceedings - 2012 7th International Conference on Availability, Reliability and Security, ARES 2012* (2012), 143–148.

[31] Thomas Würthinger, Christian Wimmer, Christian Humer, Andreas Wöß, Lukas Stadler, Chris Seaton, Gilles Duboscq, Doug Simon, and Matthias Grimmer. 2017. Practical Partial Evaluation for High-Performance Dynamic Language Runtimes. In *ACM SIGPLAN Conference on Programming Language Design and Implementation.*