Monocle: Transient Execution Proof Memory Views for Runtime Compiled Code

Matteo Oldani
ETH Zurich
Zurich, Switzerland
Oracle Labs
Zurich, Switzerland
matteo.oldani@oracle.com

William Blair
Oracle Labs
Washington D.C., USA
wdblair@bu.edu

Shweta Shinde
ETH Zurich
Zurich, Switzerland
shweta.shinde@inf.ethz.ch

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

Abstract

Most managed runtime environments abandoned attempts to mitigate speculative execution attacks between different trust domains within the same process (in-process), in favor of using process isolation to confine attackers.

Indeed, many code-generation based mitigations focused on specific variants of speculative execution vulnerabilities. This makes design-level defenses with clear guarantees, such as process isolation, a more attractive solution, despite the challenges that may arise while adopting a multi-process architecture.

We present Monocle, a fundamental runtime code-generation approach that mitigates speculative execution attacks in-process. Monocle is based on software fault isolation (SFI) and can be validated at the machine code level to ensure all potential known Spectre gadgets are covered. Benchmarks show an overhead of only 20% over the baseline, on average, and an improvement of >4.1x when compared to using memory fences, a comprehensive baseline speculative execution barrier-based mitigation strategy.

CCS Concepts

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

Keywords

Spectre, Hardware Security, In-Process Isolation

ACM Reference Format:
Matteo Oldani, William Blair, Shweta Shinde, and Matthias Neugschwandtner. 2025. Monocle: Transient Execution Proof Memory Views for Runtime Compiled Code. In *ACM Asia Conference on Computer and Communications Security (ASIA CCS '25), August 25–29, 2025, Hanoi, Vietnam.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3708821.3733887

1 Introduction

With the advent of speculative execution attacks, managed runtime environments that execute potentially untrusted code written in a higher-level language such as Java, Python or Javascript, suddenly faced an entirely new threat model: even in absence of any regular bugs in the runtime, speculative execution side channels now allow bypassing the runtime's memory safety guarantees. Meltdown-style vulnerabilities [25] are acknowledged as hardware defects when transient execution continues after fault conditions. In contrast, Spectre-style vulnerabilities [23] enable transient execution of instructions after a misprediction in the CPU pipeline, which requires a mitigation in software.

A large body of research has investigated defenses, with many focusing on particular variants such as Spectre PHT, including static analysis and formally verified approaches [7]. However, they do not quite meet the needs of Just-in-Time (JIT) compilation, which is substantial to the performance of managed runtime environments. With JIT compilation, machine code for repeatedly traversed code paths within a program implemented in a higher level programming language is generated dynamically at runtime. Attackers can leverage JIT compilation to introduce (native) code gadgets that implement speculative execution side-channel attacks at runtime.

In this paper, we investigate the challenges and opportunities specific to JIT compilation. On one hand, time-consuming static analysis or formal verification of the produced machine code hampers runtime performance, as does comprehensive use of speculative execution barriers, which flush memory and instruction caches at specific program points (i.e., relevant loads and branches) to diffuse transient execution attacks. On the other hand, access patterns of JIT compiled code are typically limited to a specific subset of the address space which permits strong restrictions on defined access ranges in memory.

Considering the browser scenario, where a Javascript engine is used to execute scripts of individual web sites, applying process isolation sacrifices density, as the separate processes require as many Javascript engines as web pages of different trust domains simultaneously displayed in the browser [32]. Also process isolation has been shown to be susceptible to attacks when components must be co-located, either due to a design or performance constraint (e.g., consolidating processes after reaching a hard limit) [1]. If process isolation is undesirable, for example because of the memory or inter-process communication overhead, mitigations are required to isolate untrusted programs in a single process (intra-process isolation).

We present MONOCLE, a compiler framework that implements secure intra-process isolation for JIT compilers. MONOCLE is based on a straightforward flavor of Software-based Fault Isolation (SFI) applied comprehensively during compilation. The access checks implemented by its machine code patterns are also executed during speculative execution, effectively mitigating speculative execution attacks. To ensure MONOCLE is working correctly, we extended the high performance Binsweep [30] binary static analyzer with an ad-hoc policy to validate the final compilation result. Being able to validate the compilation result significantly increases the overall protection level. While production runtimes are typically subject to extensive test suites to ensure their proper operation, speculative execution attacks are hard to cover by regular regression tests. In fact, real world scenarios have shown how flaws in the mitigation deployed lead to exploitable Spectre gadgets [22]. Verifying the actual machine code produced by the compiler before execution can ensure that no known gadgets are present.

We make the following contributions in this paper:

- MONOCLE, a novel and comprehensive approach to mitigate speculative execution attacks that leverage JIT compilation in managed runtimes. We implement MONOCLE on GraalVM and its JIT compiler that is used by language runtimes for Javascript, Python and Ruby among others.
- Validation of the compilation result at runtime. Our binary analysis validates that the final compilation result is fully protected by MONOCLE and is lightweight enough to be used in production alongside runtime compilation.
- An evaluation of the performance overhead of MONOCLE compared to other mitigations based on speculative execution barriers on benchmarks in multiple languages.

2 Background

2.1 Speculative Execution Attacks

Attacks relying on speculative execution have been discovered and proven possible in recent years. In 2018 Lipp et al. [25] found Meltdown and Kocher et al. [23] unveiled Spectre. Meltdown and Spectre are two classes of attacks exploiting speculative execution. In both cases, exploits using these vulnerabilities can leak arbitrary memory locations, in some cases also across address spaces. In the next paragraph, we will summarize the main versions of Spectre, following the naming suggested by Cannella et al. [5].

Spectre PHT. This initial variant of Spectre relies on poisoning the Pattern History Table (PHT) to control conditional branch prediction. In particular, the goal is to force controlled misprediction to bypass logical access control mechanisms, such as boundary checks, implemented in software with conditional branches. After a phase where the conditional branch predictor, and consequently the PHT, are mistrained, a second phase exploits speculative execution to read past a boundary check. Notably, an attacker can bypass any conditional they can (mis)train in addition to boundary checks.

Spectre BTB. This variant relies on poisoning the Branch Target Buffer (BTB) to control the location where indirect branches will land. Compared to Spectre PHT, this variant is more powerful in real-world applications since it can steer the control flow in arbitrary positions making it possible to exploit and chain together multiple gadgets already present in the code, in a Return-oriented Programming (ROP) fashion. Subsequently the same side-channel technique as before is used to extrapolate the secrets.

Spectre RSB. This variant relies on underflowing the Return Stack Buffer (RSB). The RSB can only store the last N calls, where N depends on the size of the buffer itself. However, starting from Intel Skylake, CPUs fall back to the BTB if the RSB is empty. This allows using returns to mount Spectre BTB-like attacks.

Spectre STL. This version of Spectre does not exploit control flow speculation. Instead, it exploits data speculation. CPUs can speculate on data dependencies, in particular, some loads can be executed speculatively before knowing if they are dependent on a previous store. An attacker can exploit this by speculatively skipping a store and accessing a value that was never meant to be possible to read. Later, with the help of a usual cache side-channel, the attacker infers the speculatively read value.

Mistraining. As discussed, the first three variants need to mistrain branch predictors. This mistraining can occur in four different modes [5]:

- Executing the victim branch in the victim process (same-address-space in-place).
- Executing a congruent branch in the victim process (same-address-space out-of-place). Where congruent refers to a branch that maps to the same entry of the buffer (PHT, BTB, RSB) to be poisoned.
- Executing a shadow branch in a different process (cross-address-space in-place). Where shadow refers to a branch at the same address as the branch to be mistrained, but in a different address space.
- Executing a congruent branch in a different process (cross-address-space out-of-place).

Not all the mistraining techniques are always exploitable. For example, cross-address-space attacks on managed languages, such as JavaScript, have not been explored yet to the best of our knowledge.

Side Channels. Side channels are a key component of speculative execution attacks. The microarchitectural changes that occur during the speculation windows cannot be observed directly, since the rolled-back instructions do not leave traces in the architectural state of the CPU. However, the effects of speculatively executed instructions can be observed through side channels. Most known attacks abuse cache-based side channels to leak information as in the original Spectre attack [23]. However, attacks relying on MDS-based covert channels [4, 33], port contention-based covert channels [3, 11] as well as contention on Intel CPUs' ring buffers [31] have been proven possible.

2.2 Software Fault Isolation

Software Fault Isolation was first presented in 1993 by Wahbe et al. [37] for RISC architectures. Its main purpose is the creation of

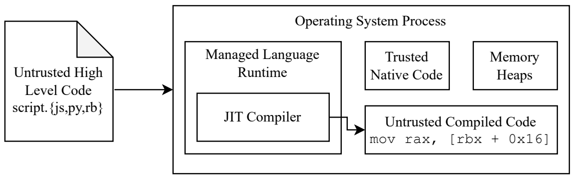

Figure 1: Our threat model describes the scenario of a managed runtime that is co-located with trusted application code in the same process. While interpretation of higher-level (i.e., memory safe) untrusted code by the managed runtime is assumed to be safe with respect to speculative execution, the addition of a JIT compiler for performance benefits can potentially allow an attacker to introduce speculative execution gadgets through JIT compiled native code.

distinct fault domains in the same address space. The goal is to allow faster interaction between domains to avoid using remote procedure calls, while strictly isolating memory so that faults in one domain cannot influence another domain. The original idea proposed to section the address space in contiguous memory regions and to restrict each memory modification to a specific domain. The restriction is achieved by masking the address to be used with a value that represents the index of the correct domain.

This seminal work has been extended to work in a CISC architecture by McCamant et al. [26].

2.3 Binsweep

Binsweep [30] is an extensible static binary analysis tool for x86 that works at the machine code level. It works under the assumption that all valid entry points in the code are marked with a specific instruction (`ENDBR64`). Binsweep uses a combination of linear sweep and recursive descent algorithms to disassemble an analyzed binary and reconstruct Control-flow Graph (CFG) from each entrypoint in the machine code. Binsweep guarantees that all runtime-reachable instructions, which conform to Binsweep's software CFI scheme, are then decoded at static analysis time. Binsweep provides an interface to implement custom security policies that are applied to the instruction stream of CFGs reconstructed from valid entrypoints.

3 Threat Model

At a high level, our threat model describes the introduction of untrusted machine code into a process by virtue of a JIT compiler that processes untrusted code supplied in a higher-level language. Figure 1 shows an overview of our threat model. The outer boundary is a process with a corresponding virtual address space. Attack scenarios that involve crossing this boundary are outside the scope of this paper. Within the process, the trusted computing base consists of the code representing the main application logic, as well as a managed runtime for a higher-level language.

We assume that the trusted computing base contains no exploitable gadgets for speculative execution attacks. This is a reasonable assumption given that static analysis tools that identify such gadgets are available [35] and code generation mitigations have been deployed in widely used compiler toolchains [6, 39]. As a result, the managed runtime is assumed to be able to securely contain interpreted execution of untrusted code with respect to speculative execution attacks.

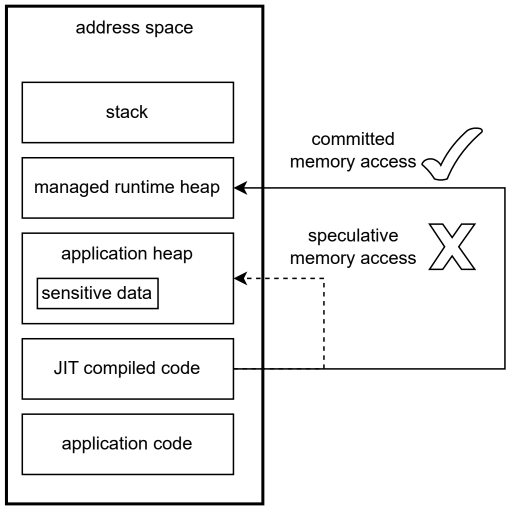

Figure 2: Both the trusted application code and the untrusted code lowered by JIT compilation to machine code share the same address space. A successful speculative execution attack creates an arbitrary memory read primitive that allows the untrusted code to access a secret in the application's heap. We assume a memory-safe runtime environment and a JIT compiler that allows rewriting untrusted code to restrict the code's access (including speculative reads) to the managed runtime heap.

The addition of a JIT compiler as part of the managed runtime introduces an additional attack surface for speculative execution attacks. Whereas the compiler itself is trusted, the machine code resulting from runtime compilation of untrusted code provided in a higher level language to the managed runtime, is not. An attacker controlling the untrusted code that is input to the JIT compiler has a degree of control over the resulting machine code that allows it to introduce speculative execution gadgets to the process. Both the trusted application code as well as the untrusted JIT compiled code share the same address space (Figure 2). Speculative execution attacks that allow the JIT compiled code to access a secret in the same address space are captured by our threat model.

An example in which the presented threat model reflects a common real scenario is a cloud-based service that embeds and runs user-provided code on the server as well as JIT compiling JavaScript provided by third-party websites in a web browser.

3.1 Scope of the Mitigation

The proposed mitigation aims at addressing speculative execution attacks belonging to the Spectre class of attacks. Specifically, we propose a mitigation for the three most known Spectre-like attacks, namely: Spectre PHT, Spectre BTB, and Spectre RSB. In subsection 6.1 we will discuss the effectiveness of the proposed mitigation also for Spectre STL, despite not being optimized for it.

Speculative accesses by untrusted code to any parts of the managed runtime heap as depicted in Figure 2 are explicitly allowed by

Table 1: A comparison of Monocle with existing approaches. Performance is graded from low (○) to high (●). Support for runtime compilation (RT), validation (Val), and mitigations against Spectre PHT, BTB, and RSB attacks are reported.

	Perf	RT	Val	PHT	BTB	RSB
Fences	○	✓	✓	✓	✓	✓
Ultimate SLH [44]	◐	✗	✗	✓	✗	✗
Venkman [34]	◐	✗	✓	✗	✓	✓
Blade [36]	●	✗	✗	✓	✗	✗
oo7 [38]	●	✗	✗	✓	✗	✗
Swivel [28]	◐	✓	✗	✓	✓	✓
Monocle	●	✓	✓	✓	✓	✓

our threat model, as the data on the managed runtime heap just captures the state of the executed untrusted code itself.

Meltdown-class attacks [25], are considered out of scope, as they are better addressed at the microarchitectural and firmware levels, and can be patched with microcode updates. We will further discuss them in section 7.

Finally all attacks relying on Spectre gadgets found in trusted code are out of scope. We assume that trusted code can be checked ahead of time as part of the development process, using, e.g., exhaustive static analysis and potential gadgets removed consequently.

3.2 Challenges and Approach

In this section, we describe the challenges posed by our threat model, in perspective with other software-based Spectre mitigations. Table 1 presents a comparable subset of mitigations extracted by Cauligi et al. [8]. As shown in this table, Monocle mitigates all the main versions of Spectre while prior approaches usually target only a subset of Spectre attacks. In addition, our threat model requires applying the mitigation at runtime, and ideally minimizing any performance overhead imposed on JIT compilation time. JIT introduces complexities not found in Ahead-of-Time (AOT) compilation, as the final binary code is generated dynamically at runtime, constraining the potential for static analysis and control during mitigation deployment. Furthermore, static analysis in this context is complicated by the inability to access a complete view of the program, as JIT compiles distinct sections of the program at different points in time.

Finally, our proposed mitigation based on compiler transformation can be verified through lightweight static analysis, as presented in subsection 4.4. Monocle combines the theoretical soundness of an SFI-based approach with a practical verification step, which guarantees the correctness of the compiled code. As a result, any potential bug in our compiler pass that limits SFI will cause static analysis to reject machine code produced during JIT compilation. Table 1 shows that validation of the mitigation is hardly found in comparable solutions.

4 Design

Figure 3 shows a high-level overview of how Monocle integrates into the JIT compilation process of a managed runtime. Untrusted code that has been executed repeatedly by the interpreter is picked

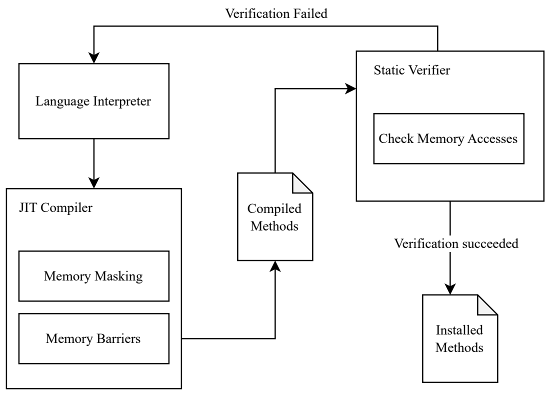

Figure 3: How Monocle integrates into the execution flow of a managed runtime: hot untrusted code is instrumented at compile time with address masking and memory barriers. The compiled code is then only installed after successfully passing custom static analysis validation to ensure the absence of speculative gadgets.

up by the JIT compiler. Monocle then applies a mixed strategy: relevant memory read accesses are instrumented to be scoped to their permitted memory region. If scoping is not possible because the permitted target region cannot be determined, Monocle falls back to using speculation barriers to ensure no speculative access that deviates from the architectural target is performed. After compilation, a static binary analysis step ensures that the mitigation has been correctly applied. On success, the compiled machine code is installed and used instead of the interpreter. If the verification fails, Monocle falls back to the interpreter. Figure 4 summarizes the memory model as well as the possible accesses to process memory. Moreover, it schematizes the proposed mitigation techniques.

4.1 Memory Model

Managed runtimes maintain a typically contiguous memory region as the runtime managed heap. This runtime managed heap contains all the objects and thus state of the untrusted code that is executed by the runtime. Heap objects are then referenced as offsets from the runtime managed heap base [17, 40]. The memory layout of the address space thus clearly separates the untrusted runtime managed heap from the trusted main application memory, which may contain sensitive data. At (JIT) compilation time, both the location of the runtime managed heap base as well as its permitted maximum size are available in well-known locations. This memory model and way of referencing heap objects enables us to use efficient scoping of memory accesses. This is in contrast to mitigation approaches developed for conventional ahead-of-time compilation such as the address or value poisoning speculative load hardening techniques implemented in LLVM [6, 44].

4.2 Address Masking

Monocle uses address masking as a primary technique to scope memory accesses. Address masking reinforces the boundary conditions for accesses to the runtime managed heap to also apply during speculative execution. With address masking, we can prevent memory accesses outside the runtime managed heap while still permitting speculative execution and thus impose significantly less performance impact compared to mitigations based on speculative execution barriers.

The technique is a reinterpretation of SFI. We use the same concept of address masking proposed by Wahbe et al. [37], however we tailor our approach to specifically mask memory accesses deemed critical from a security perspective. From a correctness point of view, the accesses are already in that memory region, indeed, we are not using SFI to create the logical separation per se. Thus, we can avoid the need for cross-region APIs based on SFI, removing a clear downside of a standard implementation of SFI.

The following pseudocode shows how a memory access is hardened using address masking:

```
// regular access
object = read(untrusted_memory_base + object_off)

// speculative execution safe access
mask = load(untrusted_memory_size)
unsafe_off = load(object_off)
safe_off = and(mask, unsafe_off)
object = read(untrusted_memory_base + safe_off)
```

The `read` instruction in this pseudocode represents a data memory access, whereas the `load` instruction does not and corresponds to accessing an immediate or value stored in a register.

The regular read of an object from the heap using base and offset is instrumented. First, the size of the runtime managed heap and the unsafe offset are loaded. Do note that the size of the runtime managed heap is a compile time constant for JIT compilation. Then the size of the runtime managed heap is used to mask the unsafe offset and scope it to the heap size, converting it to a speculative-execution safe offset. This safe offset is then used in a final step to read the object.

4.3 Memory Barriers

Despite the predominant occurrence of memory accesses within the runtime managed heap memory region in runtime-compiled code, the scope of security-critical memory accesses is more extensive. In particular, accesses beyond the confines of the runtime managed heap can be allowed, to enable tighter integration with a trusted application that requires accessing certain data structures in unmanaged memory outside the heap.

Furthermore, instances exist where the compiler cannot ascertain the location of a memory access, potentially falling outside the runtime managed heap. Masking is not possible in these cases since it will break the correctness of the program by incorrectly forcing the address onto the runtime managed heap.

To address these security-critical memory accesses that defy masking, we employ memory barriers, such as x86 `lfence`s, to block speculative execution. Our approach allows a more precise emission of memory barriers on top of memory accesses whose location does not permit masking, allowing speculation to proceed further in most cases compared to previous attempts at mitigating Spectre by extensively employing memory barriers proposed by Intel [19]. As we will further discuss in subsection 6.2, this strategy exhibits a more nuanced impact on performance.

4.4 Static Binary Analysis

Before installation of a runtime compiled method, a binary machine code analysis statically checks the security guarantees of a runtime compiled method. We based our binary analysis on the insights presented in Binsweep [30]. As reconstructing a control-flow graph is a significant challenge when analyzing machine code, the analysis assumes that all entry points to the method from the outside, as well as indirect branch targets are identified by the compiler with a specific marker instruction that is a semantic NOP when executed. This also eliminates the need for introducing heuristics to handle variable-length instructions decoding, as we can precisely trace all permissible code paths, ensuring the correctness of our decoding process.

Moreover, in our threat model we do trust the compiler, thus we can assume that bogus indirect jumps to non-marked entry points are not present in the produced JIT compiled code. As a result, we can lower the constraints around Control-flow Integrity (CFI) imposed in Binsweep, where indirect jumps and returns must be patched to follow a specific pattern that allows the tool to statically check that a given binary adheres to the CFI requirements.

Disassembling. Binsweep [30] uses two different phases, during the decoding process, to reconstruct the CFG: linear sweep and recursive descent. In the former, a byte-to-byte scan is employed to locate all the valid entry points. In the latter, disassembly from each entry point is started. This phase follows each direct branch while assuming that all the indirect branches found, at runtime, can only target a valid entry point. As already mentioned, Binsweep will enforce this through patterns on top of each indirect branch and by disabling `ret` instructions, while we can assume that all indirect branches are emitted correctly by the compiler. For this reason, we do allow return instructions.

The disassembly results are organized in a control flow graph with instruction granularity. Each instruction, among the others, holds the information regarding its class which identifies the operation code, and its operands.

Verification. The verification phase checks each memory access whose target is calculated at runtime. This involves traversing the control-flow graph and inspecting each instruction. When a relevant memory access is encountered, the verifier conducts a series of checks to determine whether to permit the access. These checks operate on three fronts: firstly, the verifier examines whether the memory access is protected by address masking; secondly, it assesses whether it is safeguarded by a memory barrier; lastly, it conducts a Spectre gadget check to ensure that the memory access cannot be exploited as part of a Spectre gadget. If none of the checks succeed, the memory access is considered unsafe. A failure in machine code verification will block the installation of the corresponding runtime compiled method, reverting the execution to an interpreted mode.

To determine if address masking is applied, we check for the presence of the masking pattern detailed in subsection 4.2 on top of the instruction being analyzed. If that check fails, then the verifier tries to locate an `lfence` preceding the critical instruction. In both scenarios, the control-flow graph facilitates the check of all the

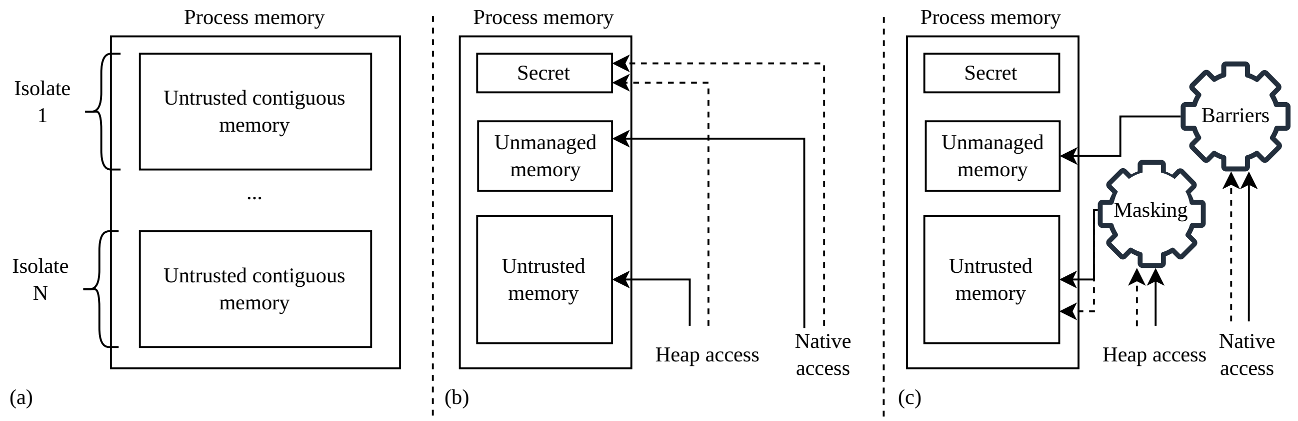

Figure 4: (a) Process memory layout for a set of untrusted runtime managed heaps in contiguous memory within a trusted process. (b) A non-mitigated scenario where memory accesses (solid lines) correctly target untrusted managed memory or a mapped region in native (i.e., unmanaged) process memory. However, speculative accesses (dashed lines) generated by both types of memory access can disclose a secret value that should not be read. (c) Our proposed mitigation in practice. Memory accesses are hardened and, as a result, speculative accesses generated from on-heap accesses are masked to untrusted memory while memory barriers disable speculation caused by native accesses.

paths leading to the critical instruction. This examination is done efficiently by backward traversal from the instruction itself.

Furthermore, within the verification process for both address masking and memory barrier checks, interleaved instructions between the anticipated pattern and the security-critical instruction are allowed. These interleaved instructions might result from optimization phases conducted by the compiler or the code scheduler during the emission of code blocks. Such instructions can be, for example, pushes that the compiler emits for register spilling as well as nops used for padding. To address this, the verifier incorporates a register tracker. This tracker differentiates between registers holding information relevant for the masking operation and those that are either unused or considered safe. The register tracker only considers entire registers, it does not distinguish between parts of a single register, i.e. rax and eax are considered the same register. The verifier considers interleaved instructions as unsafe if they touch any register used by a masking operation.

5 Implementation

We chose to implement Monocle on Oracle GraalVM 23.1 for the x86_64 architecture. GraalVM is a polyglot managed runtime environment that supports execution of multiple popular languages such as Javascript, Python and Ruby. Multiple instances of a runtime environment can be embedded into a trusted host application and execute in a single process as so-called *isolates*. An isolate captures the state of an individual runtime on a dedicated managed runtime heap. Object references are represented as offsets from the heap base ("compressed references"). For further detail on GraalVM, please refer to Appendix A.

In order to execute untrusted code, the host application creates an instance of a language runtime in the form of an isolate. This scenario reflects the setup described in our threat model.

The V8 Javascript engine uses a similar approach to support isolates and reference heap objects [17]. Objects can be referenced either through offsets from the heap base or through pointer table indirection if they reside outside the runtime managed heap. As such, Monocle could also be implemented on V8, however, we chose GraalVM for its extensive language support. Furthermore, GraalVM's graph-based intermediate representation particularly lends itself to implement Monocle (see subsection 5.3).

5.1 Address Masking

GraalVM maintains a reference to the base of the managed runtime heap in a dedicated, reserved register (r14 on x86_64). We generalize the addressing mode of x86_64, typically represented as base+(index*scale+displacement), to base+offset. Subsequently, we employ a bitwise AND operation to mask the offset, forcing it to remain within the predefined region. Notably, the heap base consistently serves as the base in the address. In the next paragraphs we present the patterns used to implement address masking. Then we will focus on the compiler changes needed to adopt the presented patterns.

Compressed References. The following snippet shows how a normal memory access using compressed references, such as mov reg, [r14 + offset], is transformed when memory masking is applied:

```
lea reg1, [offset]
mov reg2, MASK
and reg1, reg2
mov reg1, [r14 + reg1]
```

The code loads the offset of the value and then, using a mask and a bitwise and, forces the offset to be inside the isolate heap. The mask needs to be loaded separately since it can be bigger than 32 bits, making it impossible to be loaded directly as an operand of the and instruction. By default, the mask is set to 0x7FFFFFFFFF, matching the managed heap default size, however the implementation is agnostic to the mask and size of the managed heap, which is a runtime configuration.

Uncompressed References. Memory accesses can also be done using uncompressed references, i.e., regular pointers, even if the compressed references feature is enabled. In particular, the Graal compiler might optimize and decompress the reference once and later keep the uncompressed value cached in a register. Moreover, only object references that need to be stored are compressed: freshly instantiated objects can be used directly as uncompressed values. The compiler then needs to adapt the address masking pattern to accommodate this case. Uncompressed references still point to the isolate heap, thus we can derive the compressed offset starting from the uncompressed one and knowing the heap base. The following pattern shows how address masking is applied in case of uncompressed references:

```
lea reg1, [address]
sub reg1, r14
mov reg2, MASK
and reg1, reg2
mov reg1, [r14 + reg1]
```

The same considerations for the last three instructions in the case of compressed references also hold here. However, the first load does compute the full address rather than just the heap offset. Consequently, we need to subtract the isolate heap base to obtain the offset that we can later mask. In the end, a memory access to the original memory location is emitted based on the compressed addressing mode (`r14 + offsets`).

Image Heap. GraalVM native image refers to the managed runtime heap as the “image heap”. With isolates and compressed references enabled, the address of constant objects on the image heap can be loaded with a load effective address (`lea`) instruction directly combining the heap base with the given offset. This involves combining the heap base with the given offset, and the resulting address is used for retrieving the desired object. In this scenario, the masking is applied to the offset employed by the `lea` instruction, without altering the actual memory access. Additionally, the `lea` instruction incorporates the decompression mechanism, necessitating a shift of the offset by three positions due to the byte alignment of objects. The default mask now conveniently fits within the `and` instruction’s second operand. The following snippet presents the pattern used to mask image heap constant loading:

```
and reg, MASK
lea reg, [r14 + reg * 8]
mov reg, [reg]
```

Masked Memory Accesses. In both the compressed and uncompressed scenarios, we apply masking to all memory accesses that could result in a memory load, thus preventing potential information leaks. This applies universally across all x86_64 instructions dereferencing a memory location as a source operand, rather than being restricted solely to the `mov` instruction. In contrast, instructions accessing memory for the purpose of storing data are not allowed to execute out-of-order by the CPU. This precaution is taken to maintain memory coherence among different cores sharing the same caches, thereby preventing these instructions from serving as Spectre gadgets.

Based on our threat model, certain registers are considered trusted as their value is determined by the runtime and cannot be controlled by an adversary. In particular this includes the heap base register, `r14`, as well as the stack pointer, `rsp`. If only trusted registers are involved in a memory access, this access does not have to be masked. This is rarely true for accesses involving the heap base, which is always combined with an untrusted offset. However, accesses to the stack are different: the stack frame of a compiled method is set up by a trusted function prologue generated by the compiler. All stack based accesses in the compiled method are then performed using offsets that are compile-time constants that by design cannot address any data outside the current stack frame.

5.2 Compiler Extension

This section covers the implementation details of the compiler extension, which extends the Oracle GraalVM codebase by roughly 1,200 Source Lines of Code (SLoC).

Memory Accesses. The Graal compiler represents a memory location using `AddressNodes`, which serve as inputs to nodes that access memory, such as `ReadNodes`. In the low-tier stage, these nodes are transformed by the *lowering* phase to a machine-dependent representation, such as `AMD64AddressNode`.

To seamlessly incorporate the proposed mitigations, we introduce a new node type: `AMD64MaskedAddressNode`. This node takes responsibility for emitting the appropriate mitigation strategy, whether it be address masking or a memory barrier. Additionally, we extend the `AddressLoweringByNodePhase` phase to convert `AddressNodes` and their specializations into instances of the newly created node. This enhancement enables precise filtering during the phase, allowing us to discern which nodes require mitigation. We avoid transforming the memory accesses into a masked memory access for certain categories of original `AddressNode` users, in particular:

- nodes reading from reserved register (except for `r14`);
- nodes that do not use the data stored at the target location, such as `PrefetchAllocate` or `Write` nodes. These nodes, due to their nature, do not pose a risk of constructing Spectre gadgets, as they interact with memory without revealing the stored values, preventing any potential information leaks.

The newly introduced nodes are anchored on top of their users. Anchoring not only ensures correctness in the placement of safepoints for deoptimization and garbage collection but also facilitates the verification process presented in subsection 4.4. By anchoring these nodes, we ensure their proximity during code emission and block scheduling. This spatial relationship facilitates recognizing certain patterns during static analysis, enabling more straightforward and stringent assumptions. This, in turn, contributes to the overall simplicity and robustness of the static analysis process.

During emission, each `AMD64MaskedAddressNode` dynamically determines the pattern to be added to the memory access. Initially, the node checks whether a masking pattern should be emitted by inspecting the memory location identity of the base composing the address. Leveraging this information, which is normally used to check if different memory accesses might interfere with each other, allows us to identify values that must reside on the isolate heap by verifying that the current location identity is not of type `OFF_HEAP_LOCATION` or of type `GC_CARD_LOCATION`. For addresses referencing memory outside the isolate heap, the same node employs memory barriers instead of a masking pattern, ensuring that at least one of the security measures is in place.

Masking of image heap constants occurs external to the node. When such masking is required, we act on the function responsible for emitting the uncompressed reference, housing the `lea` operation detailed in subsection 5.1. If the mitigation is enabled, an additional `and` instruction is emitted to perform masking.

Furthermore, we address `FloatingReadNodes` separately. In first-tier compilation, the phase fixing those nodes is not enabled to save on compilation time. As a consequence, the masking technique cannot be applied since the nodes are floating, and it is not possible to anchor the new masked node on top of a floating node. We emit a memory barrier on top of such floating memory reads. This approach is not necessary during second-tier compilation, where all the optimizations are available.

We manually emit memory barriers in specific locations inside the compiler. For instance, `ComputedIndirectCall` directly makes use of `AMD64Address` eluding the modified lowering phase. The mitigation triggers the direct emission of an `lfence` immediately preceding the security-critical memory access. This meticulous approach ensures the timely insertion of memory barriers where required, fortifying our mitigation strategy.

Assembler Interception. After the lowering phase, which we extended to implement the core of Monocle, the compiler can still create and emit new memory reads. These accesses are directly emitted by the assembler and do not modify the compilation graph. Most of those reads are crafted during the emission of a node, as part of specific intrinsics or optimizations. Since they are created after the lowering phase, they cannot be intercepted by the modified lowering phase described above. Most of the time, those reads are non-critical, accessing the stack with a constant offset. Nonetheless, we must also check if they have to be instrumented.

To do so, we intercept operand emission to identify those operands that use an `AMD64Address`. Then, based on the values used as base and index, we decide if the address should be protected. If so, we check if any protection, either masking or fencing, has already been emitted on top of this operand. If not, then we emit an `lfence` to protect the memory access. Masking would be impractical at this stage, as during operand emission, the register allocation phase has already happened and the information needed to disambiguate on-heap vs off-heap memory accesses is no longer available. This step ensures that Monocle sees all the memory accesses ever produced, since there is no further generation of code after the assembler has emitted the operands. Any future modification to the compiler that might interfere with the masking mechanism at the lowering stage, will still be caught and protected with a memory barrier, causing a performance regression but preserving the security guarantees.

Indirect Branches. The compiler identifies basic blocks that are potential targets of indirect branches and emits an `endbr64` at their beginning, effectively marking entry points as required by the static binary analysis algorithm. This instruction is also used to identify indirect branch targets by Intel Control-flow Enforcing Technology (Intel CET) and a semantic no-operation (NOP) instruction both on CPUs with and without Intel CET support.

program	$\mathcal{G} ::= (V, E)$
V	$v ::= \textbf{node} \mid \textbf{read}\ r,\ e \mid \textbf{jmp}\ r \mid \textbf{ret} \mid \textbf{lfence} \mid \textbf{pop}\ r$
expressions	$e ::= v \mid r \mid \ell \mid e + e \mid e \,\&\, e$
locations	$\ell ::=$ memory addresses
heapbase	β
mask	χ
registers	r
values	v
trusted	$\tau : \{r\}$

Figure 5: A simplified GraalVM IR syntax for Monocle.

Based on our threat model of a managed runtime, the compiler is trusted and thus the targets of indirect branches, both forward and backward, are not under direct control by an attacker. While an attacker can still mistrain indirect branch prediction, the scope of mistraining is limited to existing entry points, which will only lead to already protected execution paths. Still, we implement indirect branch barriers (Indirect Branch Barrier (IBB)) as a defense-in-depth mechanism to address potential speculative control flow hijacks. With IBB enabled, we convert all indirect branch instructions, including function returns, to be register-indirect jumps and emit a memory barrier right before the jump instruction. The resulting forward indirect branch looks as follows:

```
mov reg, [branch_target]
lfence
jmp reg
```

Whereas returns are replaced with the following pattern:

```
pop reg
lfence
jmp reg
```

5.3 Syntax & Operational Semantics

GraalVM programs are represented in a graph based intermediate representation (IR) in static single assignment (SSA) form. Each program's IR consists of individual nodes, which may represent either program statements or values, and edges represent either control flows or data dependencies [10]. Note that this underlying IR is used for any programming language implemented on top of the Truffle Framework. That is, Truffle simplifies building programming language interpreters by generating and evaluating managed programs represented in the IR. The Native Image component compiles a programming language interpreter into an efficient executable (i.e., `graaljs`). A consequence of using this IR based approach is that Monocle's implementation, for any interpreter built on top of Truffle, can be simply defined by applying node transformations over specific nodes (i.e., memory reads) in the IR.

Figure 5 describes the fragment of GraalVM IR syntax relevant to Monocle (i.e., individual nodes that read from a memory location given by some expression). Observe that a simple walk over an untrusted program's generated graph IR can detect and transform

$$\frac{\mathcal{G}, \overline{v}, \tau \vdash v = \textbf{read}\ r,\ \beta + e\ \wedge\ \neg\, \tau(\beta)}{\mathcal{G}, \overline{v}, \tau \vdash \langle \mathcal{G}[v] = \textbf{read}\ r,\ \beta +\ e\,\&\,\chi, \overline{v} - v \rangle}\mathit{UntrustedMemoryRead}$$

$$\frac{\mathcal{G}, \overline{v}, \tau \vdash \textbf{read}\ r,\ \beta + e\ \wedge \tau(\beta)}{\mathcal{G}, \overline{v}, \tau \vdash \langle \mathcal{G}, \overline{v} - v, \tau \rangle}\mathit{TrustedMemoryRead}$$

$$\frac{\mathcal{G}, \overline{v}, \tau \vdash v = \textbf{jmp}\ \ell}{\mathcal{G}, \overline{v}, \tau \vdash \langle \mathcal{G}[v] = \textbf{mov}\ r_{target},\ \ell :: \ \textbf{lfence} :: \textbf{jmp}\ r_{target}, \tau \rangle}\mathit{Jump}$$

$$\frac{\mathcal{G}, \overline{v}, \tau \vdash v = \textbf{ret}\ \ell}{\mathcal{G}, \overline{v}, \tau \vdash \langle \mathcal{G}[v] = \textbf{pop}\ r_{target} :: \ \textbf{lfence} :: \textbf{jmp}\ r_{target}, \tau \rangle}\mathit{Return}$$

$$\frac{\mathcal{G}, \overline{v}, \tau \vdash v = \textbf{node}}{\mathcal{G}, \overline{v}, \tau \vdash \langle \mathcal{G}, \overline{v} - v, \tau \rangle}\mathit{UnrelatedNode}$$

$$\frac{\mathcal{G}, \overline{v}, \tau \vdash \overline{v} = \{\}}{\mathcal{G}, \overline{v}, \tau \vdash \mathcal{G}}\mathit{Finished}$$

Figure 6: Operational semantics for Monocle transformations.

these potentially problematic reads from a managed heap's base pointer, as well as rewrite indirect jumps and returns to insert load fences.

Figure 6 demonstrates the operational semantics of Monocle's compiler transformations as a linear scan over all the nodes in the program $\mathcal{G}$, starting with $\overline{v} = V$ (i.e., all the nodes in the graph), and continuing until all nodes have been examined. The semantics shown here represent small-step semantics (i.e., describing the transformations made to the $\mathcal{G}$ as a result of making individual node substitutions). Observe that the context during our transformation is given by $\mathcal{G}, \overline{v}, \tau$ where $\mathcal{G}$ is the current graph representation of the program at the current step, $\overline{v}$ is the set of nodes left to examine in $\mathcal{G}$, and τ is a predicate used to identify which memory regions are trusted, and hence do not require address masking. Let $\mathcal{G}[v] = v'$ represent substituting the node given by v in $\mathcal{G}$ with the node v'. After all nodes are enumerated in the graph, the final result for our transformation is $\mathcal{G}$ in the last context $\mathcal{G}, \overline{v}, \tau$. For transformations that add multiple nodes to the IR graph, let :: denote a sequence of nodes to be substituted in $\mathcal{G}$ (i.e., connecting edges between the nodes).

5.4 Static Binary Verifier

Our static binary verifier is a modified version of Binsweep [30]. We modified both the disassembly and the verification processes to suit Monocle requirements. The extension to the static binary verifier has been written in Java, relying on GraalVM Native Image to create bindings with native libraries. To implement the verification process we leveraged the extensible policy mechanism in Binsweep which allows to define properties that can be checked on the recovered CFG. As a result, we were able to express the necessary constraints required by address masking and memory fencing. The overall policy for Monocle is composed of around 1,000 SLoC.

6 Evaluation

In this section, we present both the security and performance evaluation for Monocle. subsection 6.1 provides the security analysis of the mitigation. In particular, we reason about the mitigation effectiveness w.r.t speculative execution attacks and we analyze the correctness of the implementation while showing the distribution of hardened instructions. subsection 6.2 presents the performance overheads and improvements introduced by Monocle, compared to the currently available mitigations addressing both runtime and compilation time overheads.

Table 2: Classification of critical instruction categories hardened by Monocle. The percentages refer to the fraction of each category out of the total number of critical instructions.

Hardening Categories	Average Amount
Masked	62 %
Fenced	5 %
IBB pattern	33 %

6.1 Security Evaluation

In the first paragraph of the security evaluation we analyze the correctness of the implementation. We present our verification methods and the results obtained by running our static binary verifier on the runtime compiled code. Then, in the second paragraph, we reason about the effectiveness of our proposed mitigation.

Implementation Correctness. To evaluate the correctness of the implementation, we profiled runtime compiled methods generated by Graal using our static binary analyzer. In particular, we relied on `test262`: the official ECMAScript conformance test. We collected the compiled methods resulting from the execution of the whole harness, and then we analyzed them. We confirmed that the compiled methods passed our verifier, and we collected the data regarding the amount of masking and memory barriers emitted.

The total amount of instructions were, on average, 1,435 per compiled method. Of those, an average of 98 instructions (7%) were found to be security-critical, thus relevant for our mitigation. Those critical instructions can be divided into three categories, based on the hardening method used:

- Masked instructions: instructions mitigated by address masking.
- Fenced instructions: instructions mitigated by the addition of memory barriers.
- IBB instructions: instructions that are part of an IBB pattern.

Table 2 presents the quantified categorization. As expected, the majority of security-critical instructions are hardened using masking. On the other hand, only a limited amount required fencing. Instructions belonging to IBB, despite being register-indirect memory reads, do not need hardening. In particular, the patterns are not in control of the adversary and the memory read is used to access the target location. Moreover, both paths the code can take while executing an IBB pattern lead either to an `lfence` instruction or to an `int3` instruction which will not be executed speculatively.

Spectre Mitigation Effectiveness. In the context of Spectre PHT, Monocle blocks Spectre attacks, delivering the same level of security guarantees as other techniques, such as Speculative Load Hardening (SLH). The attacker can steer the control flow of a conditional jump to either possible path, regardless of the condition. However, with our mitigation deployed, each load that the attacker may find on both paths is protected. As a result, if a load is masked then even the speculative access must be inside the runtime isolate heap. If the load is protected by a memory fence, then speculation is blocked on the path.

With a speculative control flow hijack primitive, such as Spectre BTB or Spectre RSB, an attacker can redirect the speculative control flow. However, the compiler will only produce code that can target existing entry points and therefore an adversary, who is only in control of the high-level language input to the managed runtime, can only redirect speculative control flow to code starting at valid entry points that have been considered by Monocle: All the memory accesses that can be found on a valid path are protected, as before, either by address masking or by a memory barrier. Moreover, since the set of valid entry points is known during static analysis, all the possible paths that the adversary can use are verified to be correct by our static binary analyzer.

Still, we implement IBB to mitigate speculative control flow hijacks in general, similar in spirit to the `lfence/jmp` mitigation proposed by AMD [9]. The concerns raised by Milburn et al. [27] show that the mitigation proposed by AMD might not leave enough room for speculation in a setting where SMT is enabled, however this does not apply to our threat model.

Monocle also partially mitigates Spectre STL. This Spectre variant does not rely on speculative execution along an unintended code path, but rather on speculation on data dependencies (see subsection 2.1). All memory accesses on the code path are protected either by masks or fences. Fences trivially stop speculation on data dependencies. In our masking implementation, the mask is not loaded from memory and is applied to the index register in the actual load. So assuming data dependency speculation on the access, it may fetch a stale value, but it cannot fetch data from outside the isolate heap. However, memory reads that target constant addresses outside the isolate heap remain vulnerable as those addresses are not masked.

Monocle does not close side-channels or disables speculation in general. It further does not aim to mitigate the effects of a bogus speculative load, rather it prevents any value located outside the memory region owned by an attacker from being speculatively loaded. As a consequence, our mitigation is effective regardless of the side-channel used. Since the mitigation does not aim to prevent mistraining of any predictor to begin with, it also applies in principle to cross-address-space scenarios, although those are not part of our threat model.

In comparison to speculative load hardening, Monocle's memory model of a contiguous untrusted memory region is advantageous: address masking hardens memory loads without the need to condition the load's validity to a predicate. This removes the necessity of a condition to bind the access to and allows to protect loads that are outside the speculation path triggered by conditional branches. As a result, Monocle extends beyond Spectre PHT, to which SLH is limited. Moreover, compared to value-poisoning approaches, such as LLVM's loaded value poisoning [6], Monocle acts at an earlier stage, by preventing loading protected memory to begin with rather than making the data read unusable.

Case Study. We performed a case study with the JavaScript Spectre PoC [12] published by Google's Project Zero for the V8 Javascript engine. The PoC places a secret value on V8's managed Javascript heap and leaks it with a Spectre-PHT gadget. While this setup does not match Monocle's memory model, which assumes that attacker-controlled Javascript can trivially directly access any data it places on its own heap, we can still investigate whether Monocle prevents the Spectre-PHT gadget from reading any code outside the managed heap.

To this end, we ran the PoC and compared the JIT-compiled assembly code both with and without Monocle enabled. As we show in a detailed analysis in Appendix C, with Monocle enabled all memory reads are relative to the heap base and the offset from the base is always logically constrained to be inside the managed heap. Thus, even assuming that an attacker holds a speculation primitive that allows them to bypass the array bounds check, they are unable to use this primitive to create read accesses outside the managed heap.

6.2 Performance Evaluation

We evaluate Monocle's performance across three different language implementations on top of GraalVM's Truffle framework: Javascript, Python and Ruby. For each of these we run popular benchmark suites and compare Monocle's performance against a baseline without any mitigations as well as the existing fence-based Spectre mitigations currently available in GraalVM. We also present results of microbenchmarks that compare Monocle against speculative load hardening. We further evaluate the performance impact of verifying the JIT compiled machine code at runtime.

All the experiments have been run on a machine equipped with an Intel i9-10980XE (36 cores) @ 3.000GHz CPU with turbo boost disabled and 64 GB of RAM.

Runtime overhead. We focused our tests on the JavaScript (Graal JS), Python (Graalpython), and Ruby (Truffleruby) implementations on Truffle which are deployable as native launchers that will JIT compile untrusted user-provided code. We settled for the most recent and common benchmarks publicly available, namely:

- `SunSpider`, `Octane`, and `JetStream` (1 & 2) for JavaScript;
- `PyPerformance` for Python;
- `yjit-bench` for Ruby.

Note that newer JavaScript benchmark suites incorporate some benchmarks included in older bench suites. To avoid data duplication in the benchmarks results, our results always refer to incremental versions of the named suites. Meaning, `JetStream2` results only incorporate benchmarks that were not present in `JetStream` and so on.

In all the suites, we present the slowdown of Monocle against a non-mitigated version of Oracle GraalVM. Moreover, each benchmark shows the improvements over the currently available mitigation which can be set to three different optimization levels to allow a trade-off between performance and security:

- AllTargets: All branch targets are instrumented with a barrier instruction. The exhaustiveness of this mode provides comprehensive security at the cost of significant slowdown.
- GuardTargets: Only branch targets that are relevant to memory safety are protected.
- NonDeoptGuardTargets: Further reduces the set of instrumented branch targets compared to GuardTargets by removing barriers from branches that deoptimize. Those branches

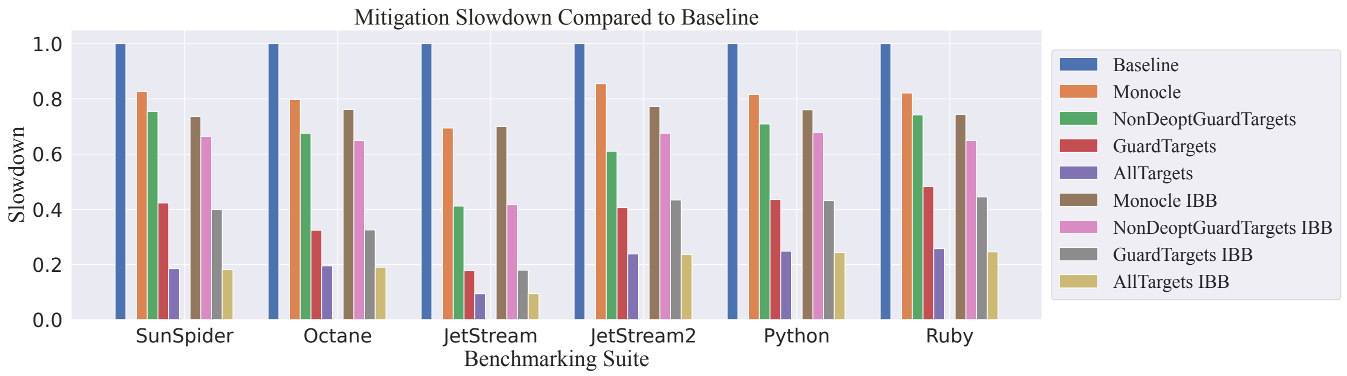

Figure 7: Slowdown of Monocle in comparison to existing mitigations in GraalVM as well as a baseline without any mitigation. Each mitigation is also paired with IBB. The evaluation is done with popular benchmark suites for Javascript, Python and Ruby.

Table 3: Monocle speedup compared to currently available Spectre mitigation on the latest version of Oracle GraalVM without IBB.

Runtime speedup without IBB				
	Monocle	NonDeopt	Guardtargets	AllTargets
SunSpider	1	1.09	1.95	4.48
Octane	1	1.80	2.44	4.07
Jetstream	1	1.68	3.86	7.32
Jetstream2	1	1.40	2.11	3.55
Python	1	1.15	1.87	3.27
Ruby	1	1.10	1.70	3.19

Table 4: Monocle speedup compared to currently available Spectre mitigations on the latest version of Oracle GraalVM with the modified software implementation of IBB.

Runtime speedup with IBB				
	Monocle	NonDeopt	Guardtargets	AllTargets
SunSpider	1	1.11	1.84	4.08
Octane	1	1.17	2.34	3.97
Jetstream	1	1.69	3.93	7.48
Jetstream2	1	1.14	1.78	3.24
Python	1	1.12	1.76	3.12
Ruby	1	1.14	1.65	2.96

can be executed only once, then control is transferred to the interpreter. Consequently, the attacker's capability of training the branch predictor is very limited. NonDeoptGuardTargets will henceforth be abbreviated with NonDeopt.

Additionally, we benchmarked the performance of Monocle and the above-mentioned mitigations with IBB. Even for this set of benchmarks, we kept the non-mitigated version of Oracle GraalVM as the baseline.

Figure 7 presents the slowdowns. We labeled *Baseline* the latest version of Oracle GraalVM. In the case of JavaScript, the benchmark measures performance as throughput, thus we computed the slowdown as T_2/T_1, where T_1 represents *Baseline* throughput and T_2 the employed mitigation throughput. For Graalpython and Truffleruby the benchmarks measured performance with time, thus the slowdown has been computed as L_1/L_2, where L_1 is the latency measured on *Baseline* and L_2 the latency with the mitigation enabled. Table 4 and Table 3 summarize the speedup of Monocle compared to the currently available mitigations with and without IBB respectively.

Despite still observing a slowdown compared to the non-mitigated version of GraalVM, the benchmarks demonstrate a major speedup in all three languages compared to all fence-based mitigations that are currently available in GraalVM. Compared to the average slowdown of the comprehensive mitigation provided by AllTargets, Monocle is 4.4x / 4.1x faster (without and with IBB).

Barrier Reduction. To evaluate how many barriers are emitted by the different mitigation approaches, we took the Spectre PoC from https://leaky.page/ and run it using `gdb` to count the number of times an `lfence` or a masking pattern (only present in Monocle) is executed. We vary the number of iterations to account for compilation tiers that are only activated after a certain number of iterations. Table 5 shows how Monocle drastically reduces the amount of memory barriers required, even compared to `NonDeoptGuardTargets`.

Table 5: Amount of masking patterns and memory barriers executed at runtime. Each cell reports the number of masked and fenced instructions in thousands.

Runtime hardened instructions [masked/fenced]				
	25k	50k	75k	100k
Monocle	365/73	740/148	1115/223	1490/298
NonDeopt	-/146	-/296	-/446	-/597
GuardTargets	-/365	-/743	-/1115	-/1493
AllTargets	-/1051	-/2129	-/3203	-/4280

Speculative Load Hardening. To compare Monocle with SLH, we created a synthetic benchmark to analyze the core patterns used by SLH, address poisoning, and Monocle, memory masking. The benchmark consists of a conditional array access that qualifies as a Spectre-PHT gadget. Since GraalVM does not offer an implementation of SLH, we chose to use the implementation of SLH in `LLVM-18` for comparison. To this end, we compile with and without SLH to obtain the machine code for the benchmark. We then manually instrument the unprotected machine code with the masking as employed by Monocle.

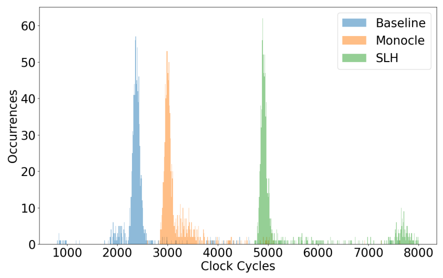

Figure 8: Execution time of the SLH benchmark as measured across 1000 iterations per setting.

Table 6: Comparison of Monocle and SLH against a baseline without any mitigations. The measurements are done with the LLVM machine code analyzer over 100 simulated iterations.

	Runtime	Iter	Instructions	Cycles	uOps
Baseline	1.00x	100	1400	310	1600
Monocle	0.78x	100	2000	436	2200
SLH	0.48x	100	2500	1006	2700

Our execution time measurement results in Table 6 show that Monocle outperforms SLH, causing a slowdown of only 22% compared to 52%. Figure 8 provides further detail on the execution time distribution of the benchmark measured over 1000 iterations.

We investigate the reason for the performance difference with the LLVM Machine Code Analyzer [1]. As expected, the tool confirms an increased backend pressure from 61.45% for Monocle to 84.79% for SLH. This is caused by the inherent data dependencies of SLH, in particular the `cmov`s used in the LLVM implementation.

More details on the benchmark, including the actual machine code evaluated, are presented in Appendix B.

Compilation Time Overhead. Monocle allows the compiled methods to be analyzed before being installed to ensure the effectiveness of the mitigation. This requires running the static binary verifier on the produced assembly code. We evaluated the overhead of this analysis comparing it with the time spent during compilation and observed a negligible overhead. In particular, we recorded the compilation times, for both Tier 1 and Tier 2 compilations of the three benchmarks suites used for the performance evaluation. We observed the following results:

- Tier 1: average compile time of 41ms and a median of 17ms;
- Tier 2: average compile time of 402ms and a median on 98ms.

Running the static binary analyzer on the same set of compiled methods shows on average overhead of 0.25ms and a median overhead of 0.20ms. We consider less than 1% performance overhead negligible, especially since it is only incurred once, during compilation.

[1] https://llvm.org/docs/CommandGuide/llvm-mca.html

7 Discussion & Related Work

Swivel [28] applies comparable Spectre mitigations to the Lucet WebAssembly compiler [2]. Unfortunately, Lucet has been discontinued years ago and was limited to WebAssembly. In contrast, Monocle is implemented and evaluated on a multi-language production runtime. Monocle also increases the overall protection level by addressing the challenge of verifying the actual machine code produced by the compiler before execution. This significantly reduces the risk of bugs in the compiler, which is important as Spectre vulnerabilities can be hardly covered by regression tests.

In SpecCFI [24] the authors propose a similar approach to mitigate Spectre BTB and Spectre RSB, using known mitigations for Spectre PHT. However, their assumption of only using hardware-assisted CFI collides with the lack of support on server operating systems. SpecCFI does not include a verification phase and, since mistraining of the BTB is still possible with CFI on, an attacker can find gadgets starting from a valid entry point. In contrast, Monocle deploys address masking on all the memory accesses to prevent an attacker from finding a valid gadget close to a valid entry point.

Ghostbusting [21] aims to enhance Spectre protection within a process, by utilizing Intel® MPK to establish separate isolation domains. However, the impracticality of relying solely on Intel® MPK arises due to its limitation to only 16 domains, making it unsuitable for isolating multiple sandbox instances. Moreover, it is only limited to CPUs supporting Intel® MPK. Finally, the approach requires changes to both the code and the underlying operating system while our approach only relies on a compiler extension. Narayan et al. [29] proposed another intra-process isolation mechanism which both improves over existing SFI techniques and mitigates Spectre for processes using the proposed sandbox. However, the approach is hardware-assisted and requires changes to the CPU microarchitecture and the operating system.

Among Intel's proposal for speculative execution control, Indirect Predictor Control (`IPRED_DIS_U`) [20] allows disabling indirect branch predictions. Enabling this feature on supported CPUs will mitigate all speculative control flow hijacks. If enabled, IBB is entirely redundant. However, disabling indirect branch prediction in hardware affects the entire process, thus presumably causing significantly higher impact on performance

An orthogonal approach for mitigating multiple Spectre variants with a comprehensive software-only solution has been proposed by Hertogh et al. [18]. In their work, they physically separate different security domains by allowing them to run on separate cores.

8 Conclusion

This paper presents Monocle, a novel approach tailored to address the challenges posed by speculative execution attacks within managed runtime environments employing JIT compilation. Monocle introduces a pragmatic yet effective strategy combining software fault isolation concepts and memory barriers. Monocle effectively mitigates speculative execution attacks, ensuring that all compiled code is statically checked to verify that potentially vulnerable memory accesses have been hardened. The proposed implementation shows significant performance improvements over the existing Spectre mitigation available in Oracle GraalVM while achieving the same level of security.

References

[1] Ayush Agarwal, Sioli O'Connell, Jason Kim, Shaked Yehezkel, Daniel Genkin, Eyal Ronen, and Yuval Yarom. 2022. Spook.js: Attacking Chrome strict site isolation via speculative execution. In *IEEE Symposium on Security and Privacy*.
[2] Bytecode Alliance. 2022. Lucet is a native WebAssembly compiler and runtime. https://github.com/bytecodealliance/lucet.
[3] Atri Bhattacharyya, Alexandra Sandulescu, Matthias Neugschwandtner, Alessandro Sorniotti, Babak Falsafi, Mathias Payer, and Anil Kurmus. 2019. SMoTherSpectre: Exploiting Speculative Execution through Port Contention. In *ACM Conference on Computer and Communications Security*.
[4] Claudio Canella, Daniel Genkin, Lukas Giner, Daniel Gruss, Moritz Lipp, Marina Minkin, Daniel Moghimi, Frank Piessens, Michael Schwarz, Berk Sunar, Jo Van Bulck, and Yuval Yarom. 2019. Fallout: Leaking Data on Meltdown-resistant CPUs. In *ACM Conference on Computer and Communications Security*.
[5] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. 2019. A Systematic Evaluation of Transient Execution Attacks and Defenses. In *USENIX Security Symposium*.
[6] Chandler Carruth. [n. d.]. Speculative Load Hardening. https://llvm.org/docs/SpeculativeLoadHardening.html.
[7] Sunjay Cauligi, Craig Disselkoen, Daniel Moghimi, Gilles Barthe, and Deian Stefan. 2022. SoK: Practical Foundations for Software Spectre Defenses. In *IEEE Symposium on Security and Privacy*.
[8] Sunjay Cauligi, Craig Disselkoen, Daniel Moghimi, Gilles Barthe, and Deian Stefan. 2022. SoK: Practical Foundations for Spectre Defenses. In *IEEE Symposium on Security and Privacy*.
[9] Advanced Micro Devices. 2023. Software Techniques for Managing Speculation on AMD Processors. https://www.amd.com/content/dam/amd/en/documents/epyc-technical-docs/tuning-guides/software-techniques-for-managing-speculation.pdf.
[10] 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 *The ACM Workshop on Virtual Machines and Intermediate Languages*.
[11] Jacob Fustos, Michael Bechtel, and Heechul Yun. 2020. SpectreRewind: Leaking Secrets to Past Instructions. In *ACM Workshop on Attacks and Solutions in Hardware Security*.
[12] Google. 2021. Spectre JavaScript PoCs. https://leaky.page/.
[13] GraalVM. 2024. FastR: Run your R code faster and more efficiently with GraalVM runtime for R. https://www.graalvm.org/r/.
[14] GraalVM. 2024. GraalJS: A high-performance embeddable JavaScript runtime for Java. https://www.graalvm.org/javascript/.
[15] GraalVM. 2024. GraalPy: A high-performance embeddable Python 3 runtime for Java. https://www.graalvm.org/python/.
[16] GraalVM. 2024. Truffleruby: Run your Ruby code faster on GraalVM's Ruby implementation. https://www.graalvm.org/ruby/.
[17] Samuel Gross. 2024. The V8 Sandbox. https://v8.dev/blog/sandbox.
[18] Mathé Hertogh, Manuel Wiesinger, Sebastian Österlund, Marius Muench, Nadav Amit, Herbert Bos, and Cristiano Giuffrida. 2023. Quarantine: Mitigating Transient Execution Attacks with Physical Domain Isolation. In *Symposium on Recent Advances in Attacks and Defenses*.
[19] Intel. 2018. Bounds Check Bypass Mitigation. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/advisory-guidance/bounds-check-bypass.html.
[20] Intel. 2024. Branch History Injection and Intra-mode Branch Target Injection. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/branch-history-injection.html.
[21] Ira Ray Jenkins, Prashant Anantharaman, Rebecca Shapiro, J. Peter Brady, Sergey Bratus, and Sean W. Smith. 2020. Ghostbusting: mitigating spectre with intraprocess memory isolation. In *Symposium on Hot Topics in the Science of Security*.
[22] Jason Kim, Stephan van Schaik, Daniel Genkin, and Yuval Yarom. 2023. iLeakage: Browser-based Timerless Speculative Execution Attacks on Apple Devices. In *ACM Conference on Computer and Communications Security*.
[23] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2019. Spectre Attacks: Exploiting Speculative Execution. In *IEEE Symposium on Security and Privacy*.
[24] Esmaeil Mohammadian Koruyeh, Shirin Haji Amin Shirazi, Khaled N. Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. 2020. SpecCFI: Mitigating Spectre Attacks using CFI Informed Speculation. In *IEEE Symposium on Security and Privacy*.
[25] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown: Reading Kernel Memory from User Space. In *USENIX Security Symposium*.
[26] Stephen McCamant and Greg Morrisett. 2005. Efficient, Verifiable Binary Sandboxing for a CISC Architecture. (2005).
[27] Alyssa Milburn, Ke Sun, and Henrique Kawakami. 2023. You Cannot Always Win the Race: Analyzing mitigations for branch target prediction attacks. In *IEEE European Symposium on Security and Privacy*.
[28] Shravan Narayan, Craig Disselkoen, Daniel Moghimi, Sunjay Cauligi, Evan Johnson, Zhao Gang, Anjo Vahldiek-Oberwagner, Ravi Sahita, Hovav Shacham, Dean Tullsen, and Deian Stefan. 2021. Swivel: Hardening WebAssembly against Spectre. In *USENIX Security Symposium*.
[29] Shravan Narayan, Tal Garfinkel, Mohammadkazem Taram, Joey Rudek, Daniel Moghimi, Evan Johnson, Chris Fallin, Anjo Vahldiek-Oberwagner, Michael LeMay, Ravi Sahita, Dean Tullsen, and Deian Stefan. 2023. Going beyond the Limits of SFI: Flexible and Secure Hardware-Assisted In-Process Isolation with HFI. In *ACM Conference on Architectural Support for Progamming Languages and Operating Systems*.
[30] Matteo Oldani, William Blair, Lukas Stadler, Zbyněk Slajchrt, and Matthias Neugschwandtner. 2024. Binsweep: Reliably Restricting Untrusted Instruction Streams with Static Binary Analysis and Control-Flow Integrity. In *ACM Cloud Computing Security Workshop*.
[31] Riccardo Paccagnella, Licheng Luo, and Christopher W. Fletcher. 2021. Lord of the Ring(s): Side Channel Attacks on the CPU On-Chip Ring Interconnect Are Practical. In *USENIX Security Symposium*.
[32] Charles Reis, Alexander Moshchuk, and Nasko Oskov. 2019. Site isolation: Process separation for web sites within the browser. In *USENIX Security Symposium*.
[33] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. 2019. ZombieLoad: Cross-Privilege-Boundary Data Sampling. In *ACM Conference on Computer and Communications Security*.
[34] Zhuojia Shen, Jie Zhou, Divya Ojha, and John Criswell. 2019. Restricting Control Flow During Speculative Execution with Venkman. *CoRR* abs/1903.10651 (2019). arXiv:1903.10651 http://arxiv.org/abs/1903.10651
[35] Synopsys Editorial Team. 2018. Detecting Spectre vulnerability exploits with static analysis. https://www.synopsys.com/blogs/software-security/detecting-spectre-vulnerability-exploits-with-static-analysis.html.
[36] Marco Vassena, Craig Disselkoen, Klaus von Gleissenthall, Sunjay Cauligi, Rami Gökhan Kıcı, Ranjit Jhala, Dean Tullsen, and Deian Stefan. 2021. Automatically eliminating speculative leaks from cryptographic code with blade. *Proceedings of the ACM on Programming Languages.* (2021).
[37] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. 1993. Efficient software-based fault isolation. In *ACM Symposium on Operating Systems Principles*.
[38] Guanhua Wang, Sudipta Chattopadhyay, Ivan Gotovchits, Tulika Mitra, and Abhik Roychoudhury. 2020. oo7: Low-overhead Defense against Spectre Attacks via Program Analysis. *IEEE Transactions on Software Engineering* (2020).
[39] Tyler Whitney, Eric Brumer, Kent Sharkey, and Hiet-Block Shayne. 2021. Visual Studio 2022 - Qspectre. https://learn.microsoft.com/en-us/cpp/build/reference/qspectre?view=msvc-170.
[40] Christian Wimmer. 2019. Isolates and Compressed References: More Flexible and Efficient Memory Management via GraalVM. https://medium.com/graalvm/isolates-and-compressed-references-more-flexible-and-efficient-memory-management-for-graalvm-a044cc50b67e.
[41] 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.* (2019).
[42] Christian Wimmer and Thomas Würthinger. 2012. Truffle: a self-optimizing runtime system. In *Annual Conference on Systems, Programming, and Applications: Software for Humanity*.
[43] Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. 2013. One VM to rule them all. In *ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*.
[44] Zhiyuan Zhang, Gilles Barthe, Chitchanok Chuengsatiansup, Peter Schwabe, and Yuval Yarom. 2023. Ultimate SLH: taking speculative load hardening to the next level. In *USENIX Security Symposium*.

A GraalVM

GraalVM [43] stands as a comprehensive suite of polyglot runtime environment technologies, developed by Oracle Labs. Central to its functionality is the Graal compiler, which offers versatile capabilities. One of its primary roles is as a replacement for the standard JIT compiler in the HotSpot Java Virtual Machine (JVM). In addition to enhancing the JVM's performance, GraalVM can produce native images [41]. These images encapsulate a heap snapshot of an application after startup, coupled with all reachable code precompiled by the GraalVM compiler. This compiled code is supported by SubstrateVM, which provides essential runtime services, such as garbage collection, typically handled by a Java VM.

A.1 Truffle

An integral aspect of GraalVM's polyglot capabilities is the incorporation of Truffle [42], a framework designed for implementing programming languages. Truffle operates by performing partial evaluation on language implementations, transforming their code into an intermediate representation that the Graal compiler can directly compile into machine code. This synergy between Truffle and the Graal compiler allows for the implementation of a wide range of languages with optimal performance. GraalVM ships with a collection of pre-built Truffle language implementations, including popular languages such as Javascript [14], Python [15], Ruby [16], and R [13]. These Truffle language implementations, when compiled into native images, can be executed either as standalone entities or embedded seamlessly within regular native code applications as shared libraries.

A.2 Isolates & Compressed References

GraalVM native image supports in-process isolation via isolates [40] to allow multiple tasks to run independently in the same address space. Each isolate has its own heap, which is disjoint from other isolates' heap. As a consequence, it is not possible to have direct references to objects across isolates. Notably, the heap base of the isolate is stored in a dedicated register. The value is updated when the thread changes the isolate in which it is running.

Besides, GraalVM supports compressed references in the context of isolates. When this feature is enabled, all references are represented using 32 bits instead of 64 bits. In particular, references are stored as offsets from the heap base of the isolate. When compressed references are enabled, each isolate has a contiguous portion of memory reserved as a heap whose size can at most be 35 bits: 32 bits of offset + 3 bits of alignment since all Java objects are byte-aligned.

B Speculative-Load Hardening Benchmark

This section expands on the comparison between MONOCLE and SLH. For the comparison we created a synthetic benchmark in C that allows us to stress-test the performance of the two different core patterns: address masking for MONOCLE and address poisoning for llvm SLH. We synthesized a simple conditional array access, which translates to a Spectre-PHT gadget. As mentioned in subsection 6.2, we compile the benchmark with `LLVM-18` with and without SLH enabled to obtain both machine code for the SLH pattern as well as a plain baseline. We manually apply the address masking technique used by MONOCLE to the basline to obtain machine code for the address masking pattern.

The following code has been compiled using clang-18, with the `-O1` option as the baseline configuration. Furthermore, the same code has been compiled using both the `-O1` and the `-mspeculative-load-hardening` flags. Note that the `-O1` option has been preferred compared to the more common `-O2` to ease the manual assembly patching. Indeed, `-O2` would already make use of loop unrolling techniques.

```
int func(char *arr, int length) {
        int value = 0;
        for (int i=0; i<ITERATIONS; i++) {
                int condition = arr[i];
                if (condition) {
                        value += arr[i];
                } else {
                        value += arr[0];
                }
        }
        return value;
}
```

Finally, the assembly generated from the baseline has been modified as follows for MONOCLE. As can be seen, a mask that covers the entire address space is used since we lack the concept of an isolate, thus we cannot restrain the access to a specific subset of the address space. However, this does not influence the overall benchmark since, even with the more narrow mask that MONOCLE uses, in a correct execution the same amount of bits (none) as for the emulated versions are modified by the mask itself. The assembly code implementing the loop logic, visible in the first two versions from the `.LBB1_3` label, has been removed for the sake of clarity since the code is identical for the three versions.

The baseline assembly:

```
        movzx   edx, byte ptr [rdi + rcx]
        test    dl, dl
        jne     .LBB1_3
        movzx   edx, byte ptr [rdi]
        jmp     .LBB1_3
```

The MONOCLE assembly:

```
        mov     r14, 0xFFFFFFFFFFFFFFFF
        and     rcx, r14
        movzx   edx, byte ptr [rdi + rcx]
        test    dl, dl
        jne     .LBB1_3
        mov     r13, rdi
        sub     r13, r15
        mov     r14, 0xFFFFFFFFFFFFFFFF
        and     r15, r14
        movzx   edx, byte ptr [r13 + r15]
        jmp     .LBB1_3
```

The version compiled with SLH:

```
        movzx   r8d, byte ptr [rdi + rsi]
        or      r8b, cl
        test    r8b, r8b
        je      .LBB2_3
        cmove   rcx, rdx
        jmp     .LBB2_4
   .LBB2_3:
        cmovne  rcx, rdx
        movzx   r8d, byte ptr [rdi]
        or      r8b, cl
```

For the benchmark, ITERATIONS has been fixed to 1000 and the program has been run 1000 times for each version.

C Google Project Zero Spectre-PHT PoC

In this section we analyze how Monocle mitigates the PoC published by Google Project Zero for the V8 engine [12]. In particular, we take a look at the generated assembly for the Spectre gadget, the code responsible for the (speculative) arbitrary read, combined with the access needed for the cache side-channel employed. The following tests are run accordingly with the threat model proposed in section 3. We rely on GraalVM isolates, while using GraalJS to execute the JavaScript code used by the PoC.

In the scope of Monocle, we only care about the return statement of the spectreGadget function in the spectre_worker.js[2] file. That line is responsible for both the speculative access inside spectreArray and the side-channel, using the result of the previous speculative access to bring a portion of the probeArray inside the cache. In the following, we present the assembly code generated by GraalJS during JIT compilation of the Spectre gadget function, when Monocle is not enabled. For simplicity, we only report the instructions that encode the last line of the function.

```
# edx contains the idx
cmp    edx,DWORD PTR [r14+rax*8+0x4]
jae    0x1834
mov    ebx,DWORD PTR [rip+0x1826]
lea    rbx,[r14+rbx*8]
mov    ebx,DWORD PTR [rbx+0x18]
test   ebx,ebx
je     0x168c
mov    ebp,DWORD PTR [rsp+0x74]
lea    rax,[r14+rax*8]
movzx  edi,dil
# spectreArray[idx]
movzx  eax,BYTE PTR [rax+rdi*1+0x8]
shrx   eax,eax,ebp
and    eax,0x1
mov    esi,eax
shl    esi,0xb
cmp    esi,DWORD PTR [r14+rbx*8+0x4]
jae    0x1698
mov    DWORD PTR [r14+r11*8+0x1c],edx
nop    WORD PTR [rax+rax*1+0x0]
cmp    DWORD PTR [r15+0x20],0x0
jne    0xe85
sub    DWORD PTR [r15+0x18],0x1
jle    0xec6
lea    rbx,[r14+rbx*8]
# probeArray[...]
movzx  eax,BYTE PTR [rbx+rsi*1+0x8]
```

As can be seen, with no mitigation deployed the attacker can potentially have control over the registers used for the read from spectreArray, leading to possible speculative bogus reads outside of the isolate running this code. However, if we look at the assembly code generated when Monocle is enabled, further below, we can see that all the memory reads are relative to the heap base (in our case, the r14 register) and that the offset from the base is always logically constrained to be inside the isolate. Thus, we conclude that even assuming that an attacker holds a speculation primitive that allows them to read past the specteArray array allocated memory, they will not be able to use this primitive to read from outside of the address space reserved for the isolate.

```
lea    rdi,[r11*8+0x4]
movabs rdx,0x7fffffff
and    rdx,rdi
xchg   ax,ax
# eax contains the idx
cmp    eax,DWORD PTR [r14+rdx*1]
jae    0x14ba
mov    edi,DWORD PTR [rip+0x15f0]
lea    rdi,[r14+rdi*8]
lea    rdi,[rdi*1+0x18]
sub    rdi,r14
movabs rdx,0x7fffffff
and    rdx,rdi
mov    edx,DWORD PTR [r14+rdx*1]
test   edx,edx
je     0x1b85
mov    ebp,DWORD PTR [rsp+0x74]
lea    r11,[r14+r11*8]
lea    rax,[r11+rax*1+0x8]
sub    rax,r14
movabs r11,0x7fffffff
and    r11,rax
# spectreArray[idx]
movzx  eax,BYTE PTR [r14+r11*1]
lea    r11,[rdx*8+0x4]
movabs rdi,0x7fffffff
and    rdi,r11
shrx   eax,eax,ebp
and    eax,0x1
mov    ebx,eax
shl    ebx,0xb
nop    DWORD PTR [rax+0x0]
cmp    ebx,DWORD PTR [r14+rdi*1]
jae    0x1b76
cmp    DWORD PTR [r15+0x20],0x0
jne    0x11e7
sub    DWORD PTR [r15+0x18],0x1
jle    0x11ad
lea    rdx,[r14+rdx*8]
lea    rax,[rdx+rbx*1+0x8]
sub    rax,r14
movabs r11,0x7fffffff
and    r11,rax
# probeArray[...]
movzx  eax,BYTE PTR [r14+r11*1]
```

[2] https://github.com/google/security-research-pocs/blob/master/spectre.js/leaky.page/templates/spectre_worker.js#L61