Empirical Security Analysis of Software-based Fault Isolation through Controlled Fault Injection

Nils Bars
CISPA Helmholtz Center for Information Security
Saarbruecken, Germany
nils.bars@cispa.de

Lukas Bernhard
Independent
Dortmund, Germany
lukas.bernhard@rub.de

Moritz Schloegel
Arizona State University
Tempe, AZ, USA
moritz.schloegel@asu.edu

Thorsten Holz
Max Planck Institute for Security and Privacy
Bochum, Germany
thorsten.holz@mpi-sp.org

Abstract

We use browsers daily to access all sorts of information. Because browsers routinely process scripts, media, and executable code from unknown sources, they form a critical security boundary between users and adversaries. A common attack vector is JavaScript, which powers complex web interactions but exposes a large attack surface due to the sheer complexity of modern JavaScript engines. To mitigate these threats, modern engines increasingly adopt software-based fault isolation (SFI). A prominent example is Google's *V8 heap sandbox*, which represents the most widely deployed SFI mechanism, protecting billions of users across all Chromium-based browsers and countless applications built on Node.js and Electron. The heap sandbox splits the address space into two parts: one part containing trusted, security-sensitive metadata, and a sandboxed heap containing memory accessible to untrusted code. On a technical level, the sandbox enforces isolation by removing raw pointers and using translation tables to resolve references to trusted objects. Consequently, an attacker cannot corrupt trusted data even with full control of the sandboxed data, unless there is a bug in how code handles data from the sandboxed heap. Despite their widespread use, such SFI mechanisms have seen surprisingly little security testing.

In this work, we propose a new testing technique that faithfully models the security boundary of modern SFI implementations. Following the SFI threat model, we assume a powerful attacker who fully controls the sandbox's memory. We implement this by instrumenting memory loads originating in the trusted domain and accessing untrusted, attacker-controlled sandbox memory. We then inject faults into the loaded data, aiming to trigger memory corruption in the *trusted domain* that processes this untrusted input. We implement our approach in a tool called SbxBrk and evaluate it on the V8 heap sandbox. In a comprehensive evaluation, we identify 19 security bugs in V8 that enable an attacker to bypass the sandbox.

CCS Concepts

• **Security and privacy → Systems security**; **Browser security**; Software and application security.

Keywords

Software-based Fault Isolation, Browser Security, Fuzzing

1 Introduction

Web browsers are the primary way to access the Internet, acting as gateways to information, communication, and online services. This central role also makes them highly attractive targets: Browsers routinely process untrusted web content, including scripts and executable code. A security vulnerability in the browser, such as a memory corruption bug, logic flaw, or improper isolation of web content, can lead to severe consequences, including data breaches, malware infections, and full system compromise. In particular, executing attacker-supplied code, such as JavaScript or WebAssembly, poses a significant security risk, since this extends the attack surface from processing untrusted *data* to executing untrusted *code*. Making matters worse, modern JavaScript engines rely on complex mechanisms like *Just-In-Time (JIT)* compilation and *On-Stack Replacement (OSR)* [4, 23], which dynamically generate optimized machine code at runtime but also introduce vulnerabilities, for instance by removing runtime checks based on flawed assumptions. This complexity creates a large and complex attack surface that is difficult to secure.

To address this issue and confine untrusted code (called *confinement problem* [33]), systems employ a technique called *Software-based Fault Isolation* (SFI) [26, 36, 39, 44, 48, 49]. This technique has been widely used in various security-sensitive applications [38, 47, 50]. SFI confines the execution of potentially malicious or buggy code to a designated *fault domain*, ensuring its effects remain isolated from the rest of the system. SFI restricts untrusted code to a specific memory region and defines a strict interface for interacting with the rest of the system. This interface defines the security boundary and is typically enforced through runtime checks. To protect web browsers, Narayan et al. proposed an SFI scheme called RLBox [38]. This framework is used by the Mozilla Firefox browser to isolate third-party libraries by executing them in a WebAssembly sandbox. Firefox has used RLBox to sandbox libraries, such as font [11] and XML parsers [13], significantly reducing the impact of vulnerabilities in these components [10]. However, unlike such libraries, JavaScript execution engines are tightly coupled to the browser and can hardly be split into atomic components that can be put into disjoint fault domains.

To tackle this problem, Google introduced the *heap sandbox* SFI-mechanism in their JavaScript and WebAssembly engine V8 [25]. Given that V8 is not only used by all Chromium-based web browsers but also by Node.js, Electron, Cloudflare's Workerd, and many other systems, this heap sandbox is arguably the most widely deployed SFI implementation in practice, protecting billions of users. The development of the heap sandbox has been driven in particular by the observation that many security bugs found in the V8 engine started as logic bugs in the JIT compiler [17, 19, 25]. Importantly, this problem cannot be mitigated by using memory-safe languages like Rust or modern hardware-backed mitigations, such as *Memory Tagging Extension* (MTE) [1, 45] and *Control-Flow Integrity* (CFI) [2, 14]. While such techniques help catch or prevent many classes of memory safety issues in general, they do not address vulnerabilities introduced by *logic errors* in JIT-compiled code. Hence, Google opted to implement an SFI scheme, the heap sandbox, to protect users even when logic bugs allow the attacker to compromise the JavaScript engine's heap. Based on the assumption that logic bugs are inherent to browsers and that objects on the heap eventually get corrupted, the memory is split into two fault domains. The first domain is the untrusted heap sandbox, which contains most heap objects accessible by the executed JavaScript and WebAssembly code, and therefore may be corrupted anytime. The second domain is the trusted domain. It contains all security-critical data structures that must be protected from tampering, such as C++ stack and heap, control flow metadata, memory management structures, and JIT-compiled code. To prevent an attacker from leveraging corrupted objects in the sandbox to escape into the trusted domain, the data structures in the untrusted region are redesigned to remove unsafe primitives such as raw pointers or large offsets. Instead, offsets relative to the sandbox base express all references within the heap sandbox. If a reference to the trusted domain is needed, it is resolved through a protected lookup table. This design ensures that code that processes malformed sandbox objects cannot be used as a vehicle to break out of the sandbox, because the necessary primitives for arbitrary memory access are absent. While the lack of raw pointers makes it harder for attackers to directly corrupt trusted memory, bugs can still occur. For instance, if a value from the untrusted domain is used without sanitization as an index into a stack-allocated array, it may still lead to corruption outside the sandbox. Consequently, it is crucial that the engine treats *all* data read from it, i.e., data crossing the *fault domain boundary*, with utmost care. Rigorous security testing is needed to ensure all such code is free of bugs that would allow an attacker to escape the sandbox. Naturally, we expect this to include *fuzzing*, a dynamic testing technique that has proven immensely effective in identifying bugs [3, 8, 42], including those in JavaScript engines [7, 29].

Unfortunately, all existing fuzzing techniques are unsuited for testing the fault domain boundary of V8's heap sandbox. Their strength lies in exposing memory safety issues in the *front-end*, i.e., the part of a system responsible for parsing and processing untrusted inputs. However, this focus on the front-end fails to test the security guarantees provided by SFI approaches, as they assume a different threat model: The attacker has already exploited a vulnerability and now has arbitrary memory access, but only within a restricted execution environment. The goal is to escape that environment and compromise the trusted domain. Consequently, a more sophisticated approach is needed.

In this paper, we present a novel fuzzing method for testing SFI implementations. We inject faults into every memory load where trusted code reads data from the untrusted domain. This approach models an attacker with complete control over the contents of the untrusted heap memory who attempts to escape the sandbox to elevate privileges. In the first step, we identify all points where data might pass the trust boundary from the untrusted heap sandbox to the trusted domain. Next, we filter out *safe* accesses where load instructions are guaranteed not to access data inside the heap sandbox. For the remaining loads, we employ inline runtime checks to determine if untrusted memory is accessed. We then inject faults by mutating values crossing the boundary, simulating adversarial input from a compromised sandbox. As a feedback loop, we use code coverage to identify faults that trigger new execution paths in the target. Overall, this approach enables us to reveal vulnerabilities in how the trust boundary is enforced by systematically corrupting data as it moves from the heap sandbox into trusted code. We implemented our approach in a prototype tool called SbxBrk and evaluated it on the V8 heap sandbox. Our fault injection-based method successfully uncovered bugs that break V8's SFI guarantees and compromise the sandbox. We found 19 security bugs in the V8 heap sandbox, including stack-based buffer overflows, use-after-free issues, and double-fetch vulnerabilities that evaded the heap sandbox. Our results show that even widely deployed SFI mechanisms can contain serious security-relevant bugs, underscoring the need for more effective testing methods to ensure the secure isolation of untrusted code.

Contributions. In summary, we make the following contributions:

- We propose the first automated testing approach that is aware of and tailored to software-based fault isolation.
- We implement our approach in a tool called SbxBrk and focus on Google Chrome's V8 JavaScript engine, which represents the most important software-based fault isolation mechanism in practice.
- We uncover 19 security bugs in V8, underlining the practical relevance of our work.

Ethical Considerations and Open Science. All vulnerabilities found have been responsibly disclosed to the Google Chrome team. Per Google's policy, details were withheld for (up to) 90 days and then made public. To foster future research on this important topic, we release our source code at https://github.com/SbxBrk.

2 Technical Background

First, we provide a brief overview of the relevant background knowledge, including software-based fault isolation, the architecture of Google's V8 engine, and the technical details of V8's SFI mechanism, the heap sandbox.

2.1 Software-based Fault Isolation

Various fault isolation techniques have been developed to contain unintended or possibly malicious behavior of applications. A well-known approach is *process-based isolation*, where each application runs in its own virtual address space, a standard mechanism supported by all major operating systems [24, 30]. This ensures that any fault, such as a crash or a vulnerability that allows arbitrary memory access, is contained within the boundaries of the faulty application's memory space. As a result, such faults do not directly impact other software running on the system. Unfortunately, such strong isolation comes with a significant performance and memory overhead. For instance, when a program uses an untrusted shared library and wants to isolate it, placing it in a separate process requires a costly inter-process communication mechanism to be employed. This reduces efficiency and increases the system's complexity, potentially introducing new bugs [16, 18, 20, 21].

To mitigate the performance impact while still enforcing isolation, *software-based fault isolation* (SFI) has emerged [26, 35, 36, 48, 50]. SFI assigns each component (e.g., a shared library or plugin) its own *fault domain*, a logically separate portion of the application's (virtual) address space [48]. To enforce the new security boundary between the *fault domains*, code is rewritten or instrumented to ensure that all memory accesses and control-flow transfers (e.g., jumps or calls) stay within their designated *fault domain*. This confines each component and prevents faults from spreading across the newly established *fault domain boundary*.

In practice, WebAssembly (Wasm) [41] is one of the most widely used SFI implementations. WebAssembly executes code inside a sandboxed environment, isolating it from the embedding application and other components. It enforces a structured memory model that prevents arbitrary memory accesses, ensuring that each code module operates within its own sandboxed environment. This is achieved through techniques such as *linear memory addressing*, which restricts memory access to a linear memory region, and *bytecode validation*, which ensures that only safe instructions are executed. Building on the isolation provided by WebAssembly, RLBox [38] enables safe execution of code written in memory-unsafe languages within the same process. Mozilla uses RLBox to compartmentalize potentially erroneous components, such as third-party libraries for parsing fonts [11] or decoding audio content [12]. By compiling the unsafe code to WebAssembly, RLBox ensures that faults are contained and that memory safety and control-flow integrity are maintained outside the sandbox. Additionally, RLBox employs a static type system to taint data to reduce the likelihood of these being used without proper sanitization.

2.2 V8 Engine

The V8 engine is a highly complex engine that uses multiple tightly interleaved interpreters and compilers to facilitate efficient execution of JavaScript and WebAssembly code.

2.2.1 *Code Generation.*

The V8 engine employs both Ahead-Of-Time (AOT) and Just-In-Time (JIT) compilation techniques to execute JavaScript and WebAssembly efficiently. In the following, we explain these techniques in more detail.

Ahead-Of-Time Compilation. AOT compilation occurs during the build process of the engine itself, where V8's core runtime, including memory management, garbage collection, and built-in object representations, is compiled from C++ using traditional compilers like Clang or GCC. This process also produces essential components such as the bytecode interpreter (Ignition), the baseline and optimizing JIT compilers (Sparkplug, Maglev, Turbofan, and Turboshaft), and support libraries, such as Torque and the Code Stub Assembler (CSA). Torque, a domain-specific language (DSL), simplifies the implementation of built-in JavaScript functions by allowing them to be written in a TypeScript-like syntax. These Torque definitions are then transpiled into CSA, a lower-level abstraction that emits optimized machine code. In summary, to instrument the AOT-compiled code, the compilation of the C++ code and the assembler used by CSA must be considered.

Just-In-Time Compilation. Initially, V8 compiles JavaScript into bytecode, which the Ignition interpreter executes using AOT compiled machine code stubs. To improve performance, the Sparkplug JIT compiler translates this bytecode into machine code stubs that are linked using control flow instructions, thus eliminating the interpreter overhead of Ignition. If the code needs to be optimized further, Maglev provides better performance, while only moderately increasing overhead for compilation. For frequently executed functions, Turbofan and Turboshaft, V8's most advanced JIT compilers, recompile the code into highly optimized machine code using techniques such as inlining and speculative optimizations. If an optimization assumption is invalidated, deoptimization reverts execution to the lowest tier. All JIT tiers ultimately rely on V8's raw assembler to emit machine code efficiently. For instrumenting JIT-compiled code, it is sufficient to instrument the machine code emitted by the assembler that is shared by all JIT tiers and also used by CSA for AOT compilation.

2.2.2 *Heap Sandbox.*

The V8 heap sandbox [47] is a lightweight, in-process fault isolation mechanism designed for Chrome's JavaScript and WebAssembly execution engine V8. Its goal is to contain the effects of memory corruption originating from untrusted JavaScript or WebAssembly code and prevent it from being leveraged to escalate privileges by corrupting security-critical objects.

To enforce this boundary, the V8 address space is split into two fault domains: a trusted domain, which holds security-sensitive objects, and an untrusted heap sandbox, which contains most heap objects accessible to JavaScript and WebAssembly code. The data structures used in the untrusted portion of the engine must be adapted so that, even if they are corrupted, they cannot be leveraged to perform arbitrary memory reads or writes. This requires eliminating certain types, such as raw pointers or large integer offsets (e.g., 64-bit offsets) that could otherwise be manipulated to reference arbitrary memory locations within the full process address space. This transformation ensures that engine code processing attacker-controlled objects—essentially untrusted bytes—cannot inadvertently cause further memory corruption. The required primitives that could enable such propagation, such as raw pointers, have been completely removed from the untrusted domain.

However, removing raw pointers necessitates introducing a new way for objects to reference one another. This is done using offsets relative to a fixed base address rather than absolute pointers. That fixed base is the heap sandbox, a dedicated region of the address space reserved for untrusted heap objects. Limiting all relative references to this controlled region, the engine can enforce strict bounds on what corrupted objects can reference.

Access to trusted objects, such as JIT-compiled code or internal engine structures, is mediated through lookup tables. These tables act as controlled indirection layers, ensuring access is explicitly validated and cannot be redirected arbitrarily through corrupted data. Through this restructuring, V8 ensures that even if memory corruption occurs within the sandbox, it cannot propagate beyond its boundary or affect the trusted parts of the engine.

```
// Get a view of the heap memory.
let memory = new DataView(new Sandbox.MemoryView(0, 0x100000000));
// Write into the heap memory
memory.setUint8(0xBBBBBBBB, 0x41)
```

Listing 1: Exemplary use of the Memory Corruption API to simulate heap corruptions within V8.

On a technical level, the V8 heap sandbox works by reserving a 1 TiB virtual memory region in which most heap objects are confined. Within this cage, objects can reference other objects only in two controlled ways: either through a 40-bit offset relative to the base of the memory cage or via an index into a trusted translation table that resolves to the target's full 64-bit address. Consequently, an attacker cannot achieve memory corruption by simply overwriting pointers since all references inside the sandbox are either constrained to point to sandboxed data within the cage or are protected by trusted lookup tables that remain outside the attacker's control. Effectively, this confines arbitrary write and read primitives to the bounds of the sandbox.

To overcome this countermeasure, an attacker must find a bug in the sandbox's implementation that allows the corruption of data *outside* the sandbox. Since the sandbox does not contain any raw pointers that can be leveraged for corruption, the goal is to find code *outside* the sandbox that processes data originating from *within* the sandboxed heap in an unsound fashion. A bug in such code may allow an attacker to achieve memory corruption that is not confined by the sandbox. Unfortunately, this implies a vast attack surface that includes, for example, all built-ins mandated by the ECMAScript specification. Such a built-in might allocate a stack buffer or a heap-allocated array (e.g., a `std::vector`). If data from the sandbox is copied into such buffers without proper size checks, an attacker can corrupt memory in the trusted domain.

2.2.3 Attacker Model. The attacker model used throughout this paper is defined by Google's guidelines [9] on what is considered a security-relevant bug in V8's heap sandbox. The attack model assumes that an attacker can execute arbitrary JavaScript and WebAssembly code and is equipped with arbitrary read and write primitives for the sandboxed heap memory. These primitives can also be used concurrently while code runs in a different thread, enabling the attacker to exploit TOCTOU and double-fetch issues (see Section 2.2.4 for additional details).

The goal is then to break the isolation by corrupting memory *outside* the sandbox boundary. To qualify as a security issue, a bug must demonstrate such an escape by causing memory corruption beyond the sandbox or, more severely, by performing a controlled memory write outside of it. In short, the security model treats the sandbox as a barrier, and any ability to breach it through memory corruption is considered a significant vulnerability.

2.2.4 Memory Corruption API. To submit a bug reproducer to Google, the V8 engine provides a JavaScript API called *Memory Corruption API* [15] that provides the capabilities described in the attacker model above. In particular, this API allows for arbitrary locations of the heap sandbox to be modified. For instance, the code displayed in Listing 1 would write the value `0x41` to the address `0xBBBBBBBB`. This API allows the implementation of proof-of-concepts (PoCs) and test cases that can be integrated into V8's test suite. To perform concurrent memory modifications (to trigger TOCTOU and double-fetch bugs), V8's Worker API [37] can spawn background threads. These threads can then use the memory corruption API to perform the corruption concurrently.

3 Design

To effectively test the V8 heap sandbox, arguably the most important real-world implementation of SFI, we need a new approach that is sensitive to the fine-grained boundary between the trusted and untrusted fault domain.

3.1 Overview

Figure 1 displays an overview of our design and shows how the different components of the target, V8, and our fuzzer interact. The heap sandbox of V8 separates the virtual address space of the JavaScript and Wasm engine into two distinct fault domains. The first domain is the *heap sandbox* that is entirely untrusted and assumed to be fully controlled by an attacker (cf. Section 2.2.3). This memory region contains most of the engine's heap objects, i.e., objects allocated by JavaScript and WebAssembly code. Corruptions in this memory region are contained in this fault domain because all references stored on this heap are relative to the sandbox's base, and pointers to external memory are resolved through lookup tables (see Section 2.2.2 for details). The second fault domain is the *trusted memory*. It contains all objects for which integrity must be maintained to ensure the security of the entire system. Both fault domains are accessed by the code that constitutes the V8 engine. Consequently, any code interacting with data in the heap sandbox is part of the attack surface. In particular, this means all of the components within V8 that generate code must be considered to ensure we cover all interactions with the heap sandbox. As described in Section 2.2.1, the code emitted by V8's compilers and interpreters can be instrumented through two key techniques: modifying the assembler and adapting the C++ code of the engine. The assembler of V8 must be extended to allow precise interception of memory accesses at the instruction level. Meanwhile, the C++ code of V8 can be instrumented using a compiler plugin that automates modifications at a higher abstraction level. Together, both modifications allow us to route all memory loads through the *Interceptor* component.

From a testing point of view, memory loads can be grouped into the following three categories: Loads of values where we can determine they are located in the trusted memory space (❶) at compile-time (of V8). For example, global variables are stored in their designated memory region and will never be on the heap. Since our attack model (cf. Section 2.2.3) assumes that data in the trusted domain is benign, we can safely ignore these loads. We term such loads *uninteresting*. If a load never accesses the heap sandbox, we can avoid all runtime costs associated with instrumenting it. Thus, maximizing the number of loads in this category is desirable.

If we cannot determine a load as *uninteresting* at compile-time, it is passed to a runtime component named *Interceptor*. This component evaluates whether the load's destination is *inside* or *outside* the sandbox. Depending on the result, it is forwarded (❷) or intercepted

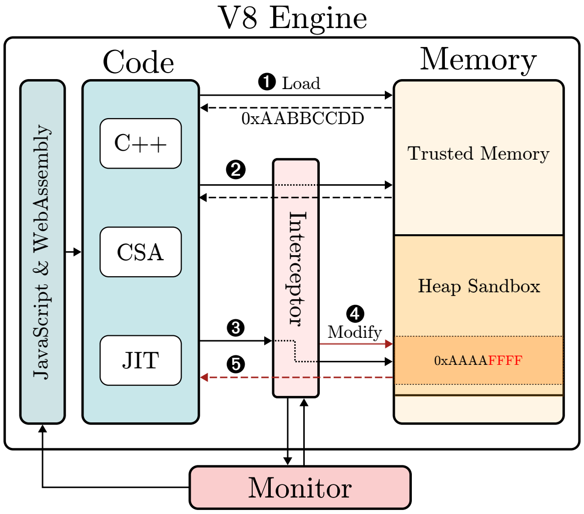

Figure 1: System overview of our prototype SbxBrk. All loads that may read from the untrusted *Heap Sandbox* are instrumented, so they pass through the *Interceptor*. The *Monitor* controls whether a fault should be injected and is responsible for providing JavaScript and WebAssembly code as input.

(❸) if it targets data *within* the heap sandbox. If we intercept a load, the *Interceptor* can mutate the memory content before loading it (❹). Such a mutation serves as our fault injection mechanism to test the code that handles data loaded from untrusted memory and will be discussed subsequently. Eventually, the load is performed and returns a (mutated) value to the engine (❺).

3.2 Fault Domain Boundary

All load instructions that access data in the untrusted heap sandbox constitute the *fault domain boundary*. Our overarching goal is to inject faults into values that cross this boundary. Following the spirit of fuzzing, we posit that any such fault injection must be sound and highly efficient. If it were unsound and we injected faults at locations that can never be under the attacker's control, the fuzzer would report false positive findings. Efficiency is crucial considering that the fault isolation mechanism isolates the heap, thus *every* memory access potentially crosses the security boundary. As these accesses are abundant, our process must be highly efficient to allow for reasonable testing throughput.

Naive Approach. A naive approach to tackle the problem would be to simply intercept every memory read access in the program and check whether it crosses the boundary at runtime. This could be implemented by leveraging the memory management unit (MMU) to raise a page fault whenever memory within the heap sandbox is accessed. However, this approach would introduce a significant slowdown for several reasons. Both reads and writes must be intercepted, as MMUs typically do not support mapping pages as write-only. Consequently, interception would have to be enforced by marking pages as no-access, triggering faults on both read and write operations. When a fault occurs, the fuzzer must temporarily grant access to the page to execute the faulting instruction. To maintain control afterwards, the fuzzer must either single-step through the instruction using a debugging interface like `ptrace` or disassemble the instruction to insert a breakpoint just after it. Once the instruction finishes, the page must be marked as no-access again to catch future accesses. These steps impose a massive performance penalty (cf. Section 5.2), making such a naive approach impractical for efficient fuzzing.

SbxBrk. Instead, we propose the following approach: SbxBrk leverages instruction-level instrumentation at compilation time, combined with a lightweight runtime component to efficiently identify and handle loads crossing the fault domain boundary. As described in Section 3.1, loads can originate from different compilers. More precisely, loads can be emitted either during the compilation of C++ code and CSA, or during runtime by the JIT compiler. We discuss each of the respective origins and how we instrument them in the following.

Loads Originating from C++. During compilation of V8, we systematically analyze each load instruction to determine whether it is guaranteed to *not* access the heap sandbox's memory and should not be considered for fault injection. We classify such load instructions as *uninteresting*. To identify the uninteresting instructions, we examine the destination of each load operation. Since the untrusted fault domain strictly encapsulates the heap, any load instruction that could be a valid target for fault injection must explicitly access heap memory. Consequently, we skip instructions loading from the stack and global variables, as these reside entirely within the trusted fault domain. We consider a load relevant for fault injection when it is impossible to conclusively show that it is not tied to heap memory after traversing all control flow and data dependencies. This procedure ensures that our approach remains sound and efficient. It filters out unnecessary fault injection targets early in the process while keeping all memory accesses that potentially cross the security boundary. Eventually, we insert a call to our *Interceptor* runtime before each *interesting* load. We pass the memory address to be read and the width of the memory access, along with the call. Based on this information, the *Interceptor* can decide at runtime whether the load targets the heap sandbox and whether faults should be injected before the memory is read.

Loads Originating from CSA & JIT. Some parts of the V8 engine are implemented by leveraging CSA, which allows the specification of low-level code using C++. During the build process of V8, a custom assembler lowers the code written using CSA into machine instructions. Fortunately, the same assembler is used when the JIT generates machine code during runtime. As a result, extending the assembler once is sufficient to cover all remaining loads not emitted as part of the C++ code.

As with the C++ code, we insert a call to the *Interceptor* before each load instruction that potentially loads data from the heap sandbox. Because the instrumentation happens on a relatively low level, we do not have access to high-level information such as control or data flow. Consequently, we cannot easily prune loads that will never access untrusted memory. Theoretically, we could move the instrumentation to a higher abstraction layer (i.e., the IR level used by the JIT) to filter loads, but this would make our approach considerably more complex. Consequently, we resort to conservatively instrumenting *all* loads on this level and leave further optimization for future work.

Interceptor. Finally, we propose to use a lightweight runtime component for all loads for which our static analysis cannot determine that they do not access heap memory. This runtime check is invoked immediately *before* each interesting load and ensures that the memory access falls within the bounds of the heap sandbox security domain. As we know the base address of the heap and its size, we can easily verify this. If the check determines an access is constrained within trusted memory, we do not consider this load for fault injection. On the other hand, if it accesses the heap sandbox contents, we can inject faults to test the security boundary. This hybrid approach—combining static elimination with dynamic validation—minimizes our overhead while allowing sound instrumentation of the security boundary.

3.3 Fault Injection

Given this precise interception of all memory accesses crossing the fault domain boundary, we can now focus on data being passed from the untrusted domain to the trusted one without being properly sanitized. We rely on *software fault injection* [5] to find such errors in trusted code handling untrusted data. More precisely, the objective is to simulate an attacker who manipulates heap data just before it is accessed by trusted code. This approach allows us to uncover vulnerabilities that arise from incorrect assumptions about the untrusted data.

To trigger these memory loads—and thus enable fault injection—the V8 engine must execute code that interacts with the sandbox. Our *Monitor* component provides this input code in the form of JavaScript files drawn from a static seed set. These files can also embed WebAssembly bytecode to cover different areas of the engine. For simplicity, we refer to these inputs as JavaScript throughout the remainder of this work, even if the JavaScript embeds WebAssembly. Importantly, these seeds are not generated or mutated by SbxBrk; instead, they must be produced up front by existing JavaScript fuzzers, such as Fuzzilli [29] or drawn from test cases included with V8.

For each fuzzing iteration, the *Monitor* provides a fuzzing input that is interpreted as a byte vector—this is the same input typically provided to a fuzzing target via a file or standard input. These bytes are then chunked into a sequence of fault injection bitmasks, with each *interesting* load consuming one such mask in the order the loads occur. Each bitmask is applied to the associated accessed memory location *before* executing the actual load; in other words, we inject a (persistent) fault at the address from which the value will be loaded. On a technical level, before executing an *interesting* load, say $load(addr, 4)$ (reading four bytes from address $addr$), our instrumentation invokes the *Interceptor*, passing address and size of the access ($interceptor(addr, 4)$). The *Interceptor* then (i) loads this value itself, $val = load(addr, 4)$, (ii) applies the bitmask $mod\_val = val \oplus bitmask$, and (iii) stores the modified value $store(mod\_val, addr, 4)$. This effectively introduces a fault to the loaded value. This way of fault injection yields two properties for the fault: *localized* and *persistent.* Using a bitmask and $\oplus$ as the operator, most parts of the original value remain unchanged; i.e., the fault is typically *localized.* In line with the fuzzing spirit, our fault injection covers the full spectrum of modifications, ranging from flipping a single bit to altering the entire value. Most injected faults involve only minor changes. The intuition is that small modifications are more likely to trigger corner cases without violating the checks enforced by the code, whereas large changes often cause the input to be discarded. At the same time, faults are also *persistent*, as we store the modified value back in memory. Once a value is mutated, any later load from the same address receives the same faulty value. The fuzzer does not need to reapply or track the mutation across the program, which reduces complexity.

A crucial consideration is *what* fuzzer input to provide to test deeper parts of the program effectively. First, we allow the bitmask to be zero, so the loaded value is not modified (since $v \oplus 0 = v$). This helps preserve the program's normal behavior, allowing SbxBrk to explore more code paths that would otherwise break if the data were corrupted too early. The stream of fuzzer input initially consists of bytes containing only zeroes, i.e., no mutations are applied. We use standard AFL++-style (non-deterministic) mutation strategies, including mutation stacking, random bit flips, and appending additional bytes to the stream. These mutations are generic and not tailored specifically to our fault injection mechanism.

To determine whether a mutation of the fuzzer input exercised novel behavior, we use *coverage feedback.* On a high level, there are three possible outcomes of our fault injection:

(1) It led to an early exit, as the program's checks considered the data to be malformed,
(2) it led to *no* observable difference in code coverage, meaning we have still explored the same code behavior, or
(3) it caused different behavior, resulting in new code coverage.

The first two outcomes are uninteresting for our purposes: Either the code checks the data safely, or our injected fault leads to no new behavior. In both cases, we do not want to keep this fault. However, when our injected fault leads to new code coverage, we can explore new program behavior. In this case, we store the fuzzing input (i.e., bitmask) for further mutation, together with the JavaScript seed file. Note that other than traditional fuzzing, *we do not mutate the JavaScript seed file.* Any set of JavaScript files (some embedding WebAssembly) can be used to execute V8 and apply our fault injection approach to loads occurring at runtime. It is desirable, however, that this set covers as much functionality of V8 as possible.

This utilization of *coverage feedback* allows us to stick to a fuzzing-esque spirit and weed out uninteresting loads and mutations at runtime. This is due to the fuzzer's inherent novelty search, striving to find more program behavior and, thus, optimizing towards interesting injected faults.

4 Implementation

Our prototype SbxBrk is based on LibAFL [28], and its core consists of roughly 5,000 lines of Rust code. For coverage instrumentation of V8, we used the AFL++ [27] compiler wrapper in version 4.22a. As a compiler for building our instrumentation pass and as a backend for AFL++, we used LLVM commit `10c6d6` of the LLVM 20 development branch. To facilitate our approach, improve performance, and enhance the bug-finding capabilities, we applied patches to V8 and LLVM. The patches and details regarding the implementation of our fuzzers are described below.

LLVM Patches. While our approach does not strictly require modifications to LLVM, we customized our LLVM version by relaxing certain ASan checks. Specifically, we removed checks for out-of-bounds (OOB) memory *reads* and for detecting overlapping memory regions passed as source and destination to `memcpy` and similar functions. These checks were disabled because only memory corruptions *outside* the heap sandbox, such as OOB *writes* or use-after-free vulnerabilities, are considered security-relevant in our threat model (cf. Section 2.2.3). Discarding uninteresting bugs reduces noise and allows the fuzzer to discover more vulnerabilities that would otherwise be obscured by early termination due to an OOB read. We stress that this is not merely a theoretical concern, as demonstrated by the bugs we identified during our evaluation (see Section 5 for details) and prior work [40].

Load Instrumentation. As described in Section 3, our instrumentation of load instructions must account for two sources of code: (1) the code generated by LLVM when compiling the C++ code of V8, and (2) the code generated by a custom assembler used during AOT compilation of V8 and JIT compilation during execution.

For loads originating from the C++ code, we use a custom LLVM pass that runs during V8's compilation. This pass iterates over all load instructions and attempts to prune those that can be provably determined not to access the heap sandbox. It recursively tracks loaded values until it reaches either an `AllocaInst` (stack allocation) or a `GlobalVariable` (global allocation). For loads that (potentially) access the heap sandbox, the pass inserts an inline check directly before the load instruction. This avoids the overhead of saving the register state and calling the *Interceptor* when the load does not touch sandboxed memory. If the check determines that the load targets the heap sandbox, control is transferred to `__fuzzer_before_heap_sandbox_load`, which receives the target address and the size of the load as arguments.

For loads emitted by the V8 assembler, we instrument low-level operations, such as `mov`, `add`, and `sub`, when their source operand references memory, e.g., `mov rax, [rbx]`. At each instrumentation site, we first save the `FLAGS` register and some scratch registers before inserting a call to `__fuzzer_before_heap_sandbox_load_preserve`. This function follows the `preserve_all` calling convention and calls our interceptor. This intermediate step ensures that no registers are clobbered, which is vital since the newly added call instruction may be executed during arbitrary function execution.

V8 Engine Patches. We modified the V8 engine to support the inline check described above. When the heap sandbox is allocated, a callback into our runtime is triggered. This callback sets the value of the `__fuzzer_heap_sandbox_base` variable, which holds the base address of the heap sandbox. This variable is exported by the runtime and used by various components, such as the LLVM pass, to quickly determine whether a load operation targets the heap sandbox.

Finally, we implemented a mechanism to pass JavaScript files to V8 via a memory buffer. This enables the *Interceptor* runtime to efficiently supply JavaScript input files without relying on traditional file system operations, thus reducing overhead.

Interceptor Runtime. The *Interceptor* runs within the V8 target process and has two primary responsibilities: (1) receiving JavaScript files and mutation byte masks via shared memory from the *Monitor* and (2) forking the V8 process and providing the newly created process with the fuzzing input received from the *Monitor* component. During execution, it intercepts loads (using the `__fuzzer_before_heap_sandbox_load` function called by our instrumentation) and potentially injects faults into memory based on the mutation byte masks. Eventually, the *Interceptor* reports the exit status of the forked process to the *Monitor*.

Delayed Fork Server. To optimize SbxBrk, we implemented a fork server that can be dynamically placed at arbitrary points during execution of V8. To mark all locations where the fork server can be placed, we introduced a new built-in `FuzzerInjectionPoint()`, which can be inserted into seed JavaScript files to indicate viable fork server positions. This primitive is especially useful for skipping the initial startup phase of the V8 engine, which involves setting up the JavaScript execution environment and triggers significant interaction with the heap sandbox. This phase is not interesting for fuzzing because an attacker would never have control during it. Additionally, bugs triggered during setup may prevent execution from reaching more diverse and test-case-specific code paths, ultimately limiting coverage.

Monitor. The *Monitor* is the central component of SbxBrk and is built on top of LibAFL [28]. It manages the fuzzing workflow by maintaining a queue of test cases that consist of JavaScript files and the associated mutation byte masks. Test cases are passed to the *Interceptor* via shared memory, which allows efficient communication. During execution, the *Monitor* tracks coverage information by reading data from the coverage map. This feedback guides the fuzzing process, helping to prioritize test cases that trigger new execution paths or uncover potential vulnerabilities.

5 Evaluation

We now evaluate the performance of our prototype SbxBrk. The evaluation is split into three parts. (i) We evaluate SbxBrk's capabilities of finding bugs compared to a modified version of Fuzzilli [29]. (ii) We perform several ablation studies to justify design decisions. (iii) Last, we demonstrate the real-world impact of SbxBrk and discuss several security bugs it found.

5.1 Setup

In the following, we detail the setup used during evaluation. The design of the conducted experiments, and their documentation, follow the recommendations by Schloegel et al. [43] and by Klees et al. [32].

Hardware Environment. We used the same hardware configuration for all our experiments: two AMD EPYC 9654 CPUs (totaling 192 physical cores / 384 logic cores), 768 GB of RAM, and an SSD as backing storage. We used CPU pinning during the experiments to restrict fuzzer instances to one (logic) core each.

Corpus. For experiments requiring a corpus of JavaScript files that may embed WebAssembly, we used Fuzzilli commit `f31876f` as a generator. The corpus was built on the hardware specified above. We launched 384 worker instances to maximize coverage and let them run for three days. Additionally, we passed the `--wasm` flag to Fuzzilli, enabling it to generate WebAssembly code inlined within the generated JavaScript code.

Target Preparation. For the preparation of V8, we used, if not noted otherwise, the build options displayed in Table 2 in the Appendix. For compiling V8, we used a different compiler depending

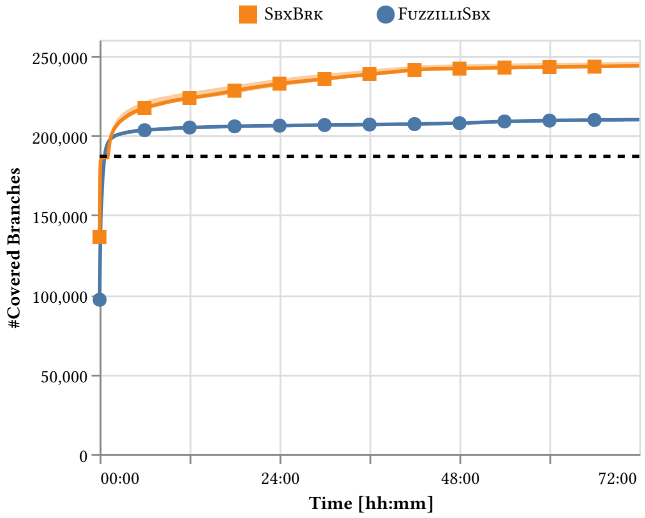

Figure 2: The coverage (in AFL++ branches) SbxBrk and FuzzilliSbx achieve over ten 3-day runs on the V8 engine. Displayed are the median run, 66% intervals, and the coverage achieved by the used seed corpus (- - -).

on the fuzzer. In case of Fuzzilli, we used the compiler bundled with V8. For SbxBrk, we relied on AFL++'s [27] compiler wrapper in version 4.22a by exporting `CC=afl-clang-fast` and `CXX=afl-clang-fast++`. Also, we set `-fpass-plugin=<path-to.so>` via `CFLAGS` and `CXXFLAGS` to enable our custom LLVM instrumentation pass. As the backing compiler for AFL++, we used LLVM `21.0.0`.

For the V8 engine itself, we used the following runtime flags: To disable certain functionality that is not fuzzing safe, i.e., that may cause a false positive, we set `--fuzzing`. Additionally, to ignore any error except heap sandbox corruptions, we use `--sandbox-fuzzing`. To increase the stability and determinism, the `--single-threaded` flag was used, such that the engine does not spawn multiple threads on its own. Furthermore, to increase the attack surface, we also passed `--allow-natives-syntax` and `--expose-gc`.

5.2 Comparison with Baseline

As we are the first to propose an approach sensitive to the security boundary introduced by software-based fault isolation, no other method is available to test V8's heap sandbox. To evaluate the capabilities of our design, we implemented a baseline fuzzer by extending Fuzzilli, the state-of-the-art JavaScript fuzzer maintained by Google [29]. We call this modified version FuzzilliSbx. It integrates support for V8's memory corruption API (see Section 2.2.4), allowing it to modify arbitrary memory locations of the heap sandbox. Integrating this API into Fuzzilli turns it into a baseline that can fuzz the heap sandbox by generating calls to the memory corruption API. We will release this patch of Fuzzilli (around 350 lines) as part of our paper's artifact. Just like for the corpora generation, we based this custom fuzzer on the Fuzzilli commit f31876f. As the target version of V8, we used commit 4715559 from December 22, 2024.

We stress that this extension of Fuzzilli is not a sophisticated fuzzer. However, due to the lack of other fault boundary-sensitive fuzzers, it can serve as a reasonable baseline to investigate whether the sandbox is simply untested, such that any fuzzer can find bugs, or if more sophisticated approaches like SbxBrk are needed.

For FuzzilliSbx and SbxBrk, we used the seed corpus described previously. Using this corpus, we executed both fuzzers for three days, ten times each. For FuzzilliSbx, we spawned 384 worker processes that synchronize their state when new inputs have been found. For SbxBrk, we executed 384 independent worker processes, simply because synchronization is currently not supported.

To measure coverage, we replayed the inputs found by both tools using the V8 binary SbxBrk used during fuzzing. We counted the number of edges covered using AFL++'s (collision-free) coverage map. Although best practice suggests using a dedicated coverage-instrumented binary [43], this was not feasible in our case: Recompiling V8 with, for example, LLVM source code coverage influences the emitted load instructions, hence some may appear in a different order or not at all, making consistent replay unreliable. To ensure fairness, we verified that our instrumentation did not expose additional coverage feedback unavailable to FuzzilliSbx. This guarantees that SbxBrk had no unfair advantage during fuzzing.

We display the coverage results of this comparative experiment in Figure 2. Both fuzzers explore many new edges beyond those already covered by the seed corpus. While FuzzilliSbx has a slight edge over our approach in the very beginning, SbxBrk quickly catches up and surpasses FuzzilliSbx, achieving 16% more coverage by the end of the three-day run. This indicates that our approach is more effective at triggering novel behavior, increasing the likelihood of uncovering bugs.

During this experiment, FuzzilliSbx found a single bug (see Table 1, #388437270) that triggered memory corruption outside the sandbox. In comparison, our prototype SbxBrk found six bugs, including the bug discovered by FuzzilliSbx. All six issues were reported to Google and have since been fixed. Table 1 provides a summary of these bugs and is discussed in detail in Section 5.4.

5.3 Ablation Studies

We conducted five ablation studies to quantify the effects of our individual design decisions and better understand the overall performance of SbxBrk.

Experiment 1: Page Fault Performance. First, we ran an experiment to determine the lower bound of the performance impact of the naive approach (see Section 3). For this approach, we propose relying on page faults to intercept loads from the heap sandbox. This evaluation helps assess the feasibility of such a naive approach. To perform this assessment, we built V8 with two different instrumentation methods:

(1) **Software-based**: Each load is instrumented by inserting callbacks, as described in this paper.
(2) **Hardware-based**: Instead of inserting a callback before a load, we insert an `int3` instruction to simulate the overhead of a load interrupted by a page fault. Additionally, we installed a signal handler that calls into the *Interceptor* as loads normally would.

We created a corpus of 100 JavaScript files by randomly selecting from the seed set used in Section 5.2. We then executed the fuzzer for 30 minutes in both configurations, ten times each. To minimize nondeterminism and keep the corpus static, we disabled coverage feedback. We also disabled mutations because the mechanism used to simulate the hardware-based approach does not expose the target address of memory loads to our runtime. As a result, executions generally take longer, since inputs are less likely to terminate early due to injected faults, which ultimately reduces the number of executions observed.

Based on the collected performance data, we observed a significant slowdown of 2.8x in terms of execution throughput for the hardware-based approach compared to the software-based version proposed in this paper. Specifically, the hardware-based configuration achieved an average of 9.2 executions per second ($\sigma = 2.09$), while the software-based version reached 25.6 executions per second ($\sigma = 2.08$). We emphasize that this slowdown represents a lower bound and is likely higher in real-world scenarios for two reasons. First, in our test, only load instructions trigger interrupts. However, in a real scenario, page faults would also occur for store instructions, as CPUs typically do not support raising faults only for read accesses. Second, to maintain precise control over each load, the fuzzer must reliably step over the faulting instruction before resuming normal execution. This can be implemented in one of two ways: by setting a breakpoint immediately after the instruction or by using `ptrace`-based single-stepping. Regardless of the method used, the mechanism involves several costly steps after each page fault: (1) Temporarily adjusting page permissions to allow the faulting instruction to complete. (2) Resuming execution. (3) Catching the resulting signal or trap (breakpoint interrupt or `ptrace` event). (4) Cleaning up: either removing the breakpoint or disabling single-stepping. (5) Restoring original page protections. (6) Resuming execution again. These additional steps introduce a significant slowdown. Unfortunately, accurately measuring this impact would require developing a completely new fuzzer, which is beyond the scope of this ablation study. Instead, we conservatively report the overhead for the underapproximation of this naive approach and show that it already causes a significant slowdown (even though it represents a lower bound).

Experiment 2: Fuzzing without coverage feedback. For classical fuzzers, relying on code coverage proved to be an efficient way to improve the performance of the fuzzer. However, the approach presented in this paper is different from most other fuzzers. Therefore, whether it can benefit from using coverage as feedback is uncertain. To investigate this, we performed an ablation by disabling the coverage feedback for SbxBrk and otherwise performing the same experiment as described in Section 5.2, running the fuzzer ten times for three days and comparing the number of bugs found afterwards.

During this experiment, the fuzzer without feedback could not uncover *any* security-relevant bug. It identified only out-of-bounds reads and stack exhaustion bugs that were also found when using coverage. These bugs are irrelevant in our attacker model, and Google does not fix them (e.g., #384486462). Thus, they are easily triggered. This struggle to find security-relevant bugs indicates that coverage is crucial for the success of our technique.

Experiment 3: Load Pruning. We use static analysis during load instruction instrumentation to avoid instrumenting loads that are guaranteed not to access the heap sandbox. To measure the impact of this optimization, we ran an ablation study that compared two builds of V8 (`eeb2845`): one with load pruning enabled and one without. When load pruning is disabled, all load instructions are instrumented. Each load is checked at runtime to see if it accesses the heap sandbox. With load pruning enabled, this check is omitted for loads that can be ruled out statically, avoiding the associated runtime cost. To keep results consistent, we disabled coverage feedback and used a fixed seed corpus, as in the *Page Fault Performance* experiment, reducing the nondeterminism throughout the execution. We executed each configuration for 30 minutes and repeated the experiment 10 times each.

Despite load pruning eliminating 52% of the instrumented loads (from $1,430,287$ down to $742,225$), the performance gain was modest: Execution speed increased by only 3.53 execs/s on average, from 70.62 ($\sigma = 1.08$) to 74.15 ($\sigma = 1.29$) execs/s. This equates to a performance increase of 5%. This small gain suggests that our runtime check is already highly efficient.

Experiment 4: Compiler Effects on Loads. The heap sandbox security boundary is defined by the loads that access the sandboxed memory. While C++ defines these loads on a high level, the compiler decides *whether* and *when* a load happens on the machine code level. This can lead to various problems affecting the heap sandbox's security. For example, depending on the compiler and the flags used, the compiler may choose to either temporarily keep a value on the stack and write it to the heap at the end of the function, or to store the value directly on the heap and load it back from the heap if needed later. In the former case, the function does not consume any attacker-controlled data, while in the latter case, the attacker can manipulate the on-heap data before it is read back. Consequently, bugs may only get visible if the compiler is updated or the build flags are changed.

Ideally, we could automatically determine if a change in the code emitted by the compiler introduces a new bug, but this is a challenging problem in practice. To understand the extent and impact of using different compiler versions, we use a proxy metric: the difference in the number of store and load instructions emitted per line of code for different compiler versions. We used LLVM versions `10c6d63` (July 2024) and `3bd3e06` (January 2025) to build V8 (`eeb2845`). For each of these builds, we used a custom LLVM pass that records a mapping of the source code location to the number of load instructions emitted. Based on these values, we can approximate the fraction of source code lines affected by different compiler versions. Notably, not all changes in the number of loads necessarily indicate a bug, and this approach is also incomplete, as the order and precise location of loads are crucial.

Across both builds, $260,434$ unique source code lines emitted at least one load. From these source code lines, 2.84% changed the number of loads emitted if the LLVM compiler version is changed. This shows that the fault domain boundary is not static and that using two compiler versions only six months apart may change whether a bug can be triggered or not. Our results in Section 5.4 confirm that these effects are not just theoretical, as we discovered bugs that only surface due to differences in how the compiler lowers high-level C++ into machine code.

Experiment 5: Delayed Fork Server. As part of our implementation, we introduced a technique that delays the fork server's start until after the target's initialization. The underlying hypothesis is that this improves fuzzing effectiveness by skipping the V8 engine's start-up phase and skipping parts of the input JavaScript file, allowing the engine to be fuzzed in a more targeted fashion. This matters because injected faults may crash the engine early, reducing the likelihood that later parts of the code are ever reached. By delaying the fork server, we increase the probability that code at the end of the input will be executed.

To evaluate the impact of this feature, we repeated the bug experiment described in Section 5.2, but with the delayed fork server disabled. Over three days, this configuration failed to uncover *any* bug. Further manual analysis shows that the engine's initialization phase is the main bottleneck. During start-up, the V8 engine creates many JavaScript objects to expose the complete set of APIs mandated by the JavaScript specification. Benchmarking this phase reveals that the fuzzer intercepts 193, 639 heap sandbox loads—consuming 737.89 KiB of the mutation mask—before the first line of JavaScript input is executed. This overhead makes it much more likely for the fuzzer to get stuck during initialization, which our delayed fork server approach avoids.

5.4 Finding New Vulnerabilities

To evaluate the capability of SbxBrk to find novel bugs, we performed a bug experiment by fuzzing V8 for several days. During this campaign, we uncovered 10 bugs beyond those reported in Section 5.2, totaling 19 bugs. These include heap out-of-bounds writes, stack-buffer overflows, and use-after-free vulnerabilities. Each bug, including a description, is listed in Table 1. Besides the listed bugs, our fuzzer found multiple out-of-bound reads that caused a segmentation fault or stack overflow. However, according to Google (e.g., #384486462), these are not considered relevant to security and are usually not fixed. We refrain from reporting such bugs to avoid overloading the developers with undesired reports. In the remainder of this section, we present several particularly interesting bugs in more detail and discuss what makes them unique to SFI.

Case Study 1: Double Fetch. One category of bugs SbxBrk found are *double fetch* bugs that occur if a value is read multiple times from the heap sandbox, and the program implicitly assumes the value did not change between reads. Bugs that we found in this category include #396446145, #392938085, and #385775375. While it is usually sufficient to treat values read from the heap sandbox as untrusted in a local context, these bugs require the developer to consider situations where constraints that held only a few lines before are unexpectedly obsolete. Notably, exploiting such bugs requires attacker-controlled code to run *concurrently* if the two loads are not naturally interleaved with code under the attacker's control. However, this condition is easily met under the assumed attack model (see Section 2.2.3).

We now take a closer look at one representative example of bugs of this type: Bug #385775375 is located in the `TypedArraySortFast` function, which is used in multiple locations to sort typed arrays. The (pseudo) code containing the bug is shown in Listing 2. The problem here is that the size of the array, which is sorted, is fetched in two locations. First, the array length (in *number of elements*) is stored in `length` at the beginning of the function. Next, the content of the array is copied into a newly allocated buffer (1). This time, `bytes` is used to determine the allocation size *in bytes*. Notably, this value might not match the one stored in the `length` variable, as it was fetched from the heap again. Eventually, `std::sort` is called (2), and the address of the last element is calculated using `length`. Consequently, `std::sort` can be tricked into sorting a larger array than allocated. Since the buffer is allocated on the trusted C++ heap, this will lead to memory corruption outside the heap sandbox.

```
RUNTIME_FUNCTION(Runtime_TypedArraySortFast) {
  <..>
  size_t length = array->GetLength();
  DirectHandle<JSArrayBuffer> buffer(<...>);
  const bool copy_data = buffer->is_shared();
  // Buffer on trusted C++ heap.
  std::vector<uint8_t> offheap_copy;
  void* data_copy_ptr = nullptr;
  if (copy_data) {
    const size_t bytes = array->GetByteLength();
    if (...) {...} } else {
      // (1) Set the size of the buffer based on `bytes`
      offheap_copy.resize(bytes);
      data_copy_ptr = &offheap_copy[0];
    }
    base::Relaxed_Memcpy(data_copy_ptr, array->DataPtr(), bytes);
  }
  ctype* data = copy_data ? <...>
  // (2) Sort the array using `length` to calculate the array end
  std::sort(data, data + length);
  <...>
}
```

Listing 2: Pseudo code containing bug #385775375. If the to-be-sorted buffer is shared (`copy_data == true`), the sorted data is copied into a new buffer that is allocated in the trusted C++ heap using `bytes` as size (1). After copying the data, it is sorted (2) using `length` to calculate the last element.

```
MemoryChunk::MemoryChunk(<...>): <...>,
  // (1) Write the index to `metadata_index`.
  metadata_index(MetadataTableIndex(address()))
{
  MemoryChunkMetadata** metadata_pointers = MetadataTableAddress();
  // (2) Use the metadata_index as index into
  // the `metadata_pointers` table.
  metadata_pointers[metadata_index] = metadata;
}
```

Listing 3: Shortened code that contains bug #389713719. First, the constructor of `MemoryChunk` initializes the `metadata_index_` on-heap member variable (1). Next, this member is used to index the `metadata_pointer_table` table without any sanitization (2).

Case Study 2: Compilation Dependent Bug. The bug reported in #389713719 is particularly interesting because it demonstrates that, as discussed in Section 5.3, bugs may depend on how the compiler lowers the high-level C++ code to machine code. The code that contains this bug is displayed in Listing 3.

The code constructs a `MemoryChunk` object inside the heap sandbox at some specific address (`address()`). Since these objects contain sensitive data that must always be of integrity, the `MemoryChunk` object only stores an integer index `metadata_index` in the heap sandbox (1). If sensitive data needs to be accessed, this variable is used as an index into the `metadata_pointers` that resolves to a `MemoryChunkMetadata` object stored in trusted memory (2).

The issue in Listing 3 was found during a fuzzing run using a V8 build configuration without optimization (i.e., compiled with `-O0`). In this configuration, an attacker may alter the value of `metadata_index` before it is used to access the lookup table, resulting in a

Table 1: All 19 security issues we found in V8 during our evaluation. Bugs are either fixed by upstream or marked as duplicate.

Bug ID	Status	Description
#385775375	fixed	`Runtime_TypedArraySortFast` is susceptible to a double fetch attack that allows an attacker to execute `std::sort` on an undersized buffer, resulting in out-of-bounds writes.
#388193955*	duplicate	Passing an unusually large value to `toExponential()` causes a stack-buffer-overflow.
#388437270*	fixed	Calling `JSON.stringify` on a String value with a length of `0xffffffff` causes an integer overflow during serialization and, eventually, an out-of-bounds write.
#388616182*	fixed	The code path for converting a `Number` to a `String` with a particular radix via `toString(16)` contains a double fetch bug. This is triggered by changing the `Number` to NaN after being checked not to be NaN, a value the subsequent code does not expect.
#389713719	fixed	When constructing a `MemoryChunk`, an on-heap pointer table index is used unsanitized to compute an address that is written to.
#389970331*	fixed	Converting a String of length `0xffffffff` via `BigInt()` causes an integer overflow and a subsequent stack-buffer-overflow.
#390205877	fixed	A variant of the `CachedTieringDecision` enum is constructed from on-heap data that causes undefined behavior in the functions that use a non-exhaustive switch statement to compute their return value.
#390453039	fixed	A variant of the `AddressType` enum is constructed from on-heap data that causes undefined behavior in the functions that use a non-exhaustive switch statement to compute their return value.
#390568183	fixed	A variant of the `MessageTemplate` enum is constructed from on-heap data that causes undefined behavior in the functions that use a non-exhaustive switch statement to compute their return value.
#392541992	fixed	Serializing a `URIError` containing a `String` with a specific length via `d8.serializer.serialize` causes a use-after-free bug.
#392938085	fixed	During parsing of the AST, one byte strings, e.g., `"A"`, are converted into `String` instances via a lookup table. This lookup process is subject to a double fetch bug that causes an out-of-bounds write.
#393989622	fixed	Comparing two strings using `localeCompare()` causes an out-of-bounds write for long strings.
#395029283*	fixed	Converting a `Number` to a `String` via `toPrecision` causes a buffer-overflow if a temporarily allocated `Number` object is mutated.
#396446145*	fixed	Inserting escape sequences into a `String` while it is processed by `JSON.parse` causes a stack-buffer-overflow.
#397875195	duplicate	Construction of a `FeedbackMetadata` enum variant causes an out-of-bounds write if converted to a `String`.
#398773898	fixed	Calling `JSON.stringify` on an unusually long property key may cause an out-of-bounds write.
#403372467	fixed	Integer overflow in the icu library when formatting a malformed `Number` causes an out-of-bounds write.
#411598604	fixed	Use-after-free when performing tear down involving large pages.
#414831374	fixed	An integer overflow in `Module::GetModuleNamespace` causes an allocation to be smaller than expected, resulting in an out-of-bound write.

*: Bug has been found as part of the experiments performed in Section 5.2, i.e., within three days.

controlled write of the value `metadata`. Crucially, while the compiler must store the value `metadata_index` to the heap—since this is what the source code expresses—it is left to the compiler's discretion if it loads the value from the heap for subsequent use, or if it keeps a copy in a register or on the stack while the function is executed. No data will likely be read from the heap if the code is compiled with optimizations. Therefore, the function does not consume any attacker-controlled data. However, when the optimization level is set to `-O0`, the compiler stores the index in the heap and loads it back a few instructions later, allowing an attacker to exploit this bug. Even though we may assume that unoptimized code is unlikely to be used in practice, it is important to fix such bugs. Otherwise, they may surface later due to using a different compiler or changes in the code that force the compiler to spill values to the heap that have been kept in a register before.

Case Study 3: Unknown Enum Variants. Unlike the typical attacker models used for traditional software fuzzing, we assume an attacker has complete control of (untrusted) heap memory. This introduces a new issue: `enum` discriminants stored on the heap must be treated as having arbitrary values. We discovered multiple bugs—such as #390205877, #390453039, #390568183, and #397875195—that exploit this by constructing `enum` variants that are not defined in the source code. This bug class appears to be previously unknown, as an umbrella bug (#390617721) for hardening `enum` variant construction was created after we reported these issues.

Although constructing unknown `enum` variants is generally allowed, problems arise when such variants reach a sink that assumes only explicitly defined variants exist. All bugs of this type led to undefined behavior in non-void functions that handled all explicitly defined `enum` variants but lacked a default case. As a result, these functions did not execute any switch case and reached a point where no return statement was executed, which is undefined behavior when the function has a non-void return type.

Case Study 4: ASan-shadowed Bugs. Unlike traditional fuzzing scenarios, our attacker model of the heap sandbox does not consider out-of-bounds reads as a bug (cf. Section 2.2.3). ASan considers such reads a bug by default, terminating the execution. However, we found that in some instances, such as #389970331 and #385775375, an out-of-bounds read is followed by an out-of-bounds *write*, which constitutes a bug in our attacker model. This demonstrates that it would be unwise to directly discard out-of-bounds reads since they can sometimes shadow more powerful primitives. Instead, we tailor ASan so that it does not terminate after encountering an out-of-bounds read.

6 Discussion

We now discuss directions for further research and limitations of our approach.

Applicability to Other SFI Tools. Although SbxBrk is designed specifically for V8, the core concept of instrumenting the fault isolation boundary can be applied to other Software Fault Isolation (SFI) implementations. For example, Firefox's RLBox [38] isolates third-party libraries by marking data that crosses from an untrusted domain into a trusted one using annotations. These annotations define the fault isolation boundary, making it possible to insert instrumentation that a fuzzer could use to inject faults *before* the data is processed by the trusted component.

Realism of our Fault Injection Model. At first glance, our ability to arbitrarily modify heap sandboxed data before every intercepted load may appear overly powerful. This might seem unrealistic, as it assumes the attacker can control memory between two loads, even if execution does not return to their JavaScript code in between. However, this scenario is realistic in practice: Using the `Worker` API, an attacker can spawn background threads that run concurrently with trusted engine code. These threads can modify heap memory concurrently, effectively enabling the interleaving that our fault injection model relies on (cf. 2.2.4). This is not just theoretical; we successfully triggered double-fetch bugs using this technique, showing that our model reflects realistic exploitation strategies (see Section 5.4).

7 Related Work

In the following, we discuss work related to our approach.

Software-based Fault Isolation Schemes. This first SFI scheme was proposed by Wahbe et al. [48] in 1993. Their approach uses a segment identifier (i.e., some prefix bits of an address) to split the address space into multiple segments. Two of these segments, one for code, the other for stack and heap, facilitate a so-called *fault domain*. All stores and control transfer instructions that can not be proven to be within their corresponding segment are termed unsafe instructions and are instrumented. McCamant et al. [35] then proposed an extension, as the prior scheme can be easily bypassed on CISC architectures, such as x86, where the variable-length encoding of instructions is used. To avoid SFI checks being bypassed, McCamant et al. propose to enforce a specific alignment and use fixed-size chunks of instructions.

Picking up this idea, Google implemented *Native Client* [50], which was used to execute untrusted x86 code in a browser context. In addition, bounds checks of data accesses can happen hardware-accelerated by relying on 80386 segmented memory [22]. Eventually, this technique was superseded by WebAssembly [41], a language specifically designed to compile code into a low-level format that can be executed without necessarily trusting the source of the code. This is enforced by executing WebAssembly via a dedicated interpreter.

Based on WebAssembly, Narayan et al. developed RLBox [38], which compiles potentially unsafe code, such as XML parsing libraries [13], into WebAssembly. Then, it runs the code in a fault-isolated environment within the same virtual address space. This system was adopted by Mozilla for Firefox to run third-party libraries that are written in C/C++. This way, Firefox cannot be compromised by bugs in these libraries. However, despite the critical importance of correctness in SFI schemes, they have received little to no rigorous testing. Our work introduces an approach to enable fuzz testing of this security boundary.

Fault Injection. Prior work has already explored *fault injection* in the context of fuzzing, either to test error handling code [31, 34, 46] or to mutate programs to produce slightly invalid output for the fuzz target [5, 6]. Both use cases differ significantly from our approach, as they target a regular fuzzing scenario and focus primarily on improving error-handling code or generating more suitable program input. Our work focuses on the security boundary imposed by SFI within a single process, and we use fault injection to model the arbitrary control of an attacker within the sandbox.

8 Conclusion

Software-based fault isolation represents a key technique for enforcing a security boundary that helps secure high-risk applications, such as web browsers that execute attacker-controlled code. Despite their critical importance in protecting end users, SFI schemes like V8's heap sandbox have received surprisingly little attention from security researchers and practitioners. In this work, we explored why existing off-the-shelf fuzzers fail to thoroughly test SFI mechanisms and introduced a novel fault injection-based method to address this gap. Our fuzzer, called SbxBrk, is the first tool to be aware of this intra-process security boundary and to model an attacker's control over the sandboxed memory faithfully. Using SbxBrk, we uncovered 19 previously unknown security bugs in Google's V8 that traditional fuzzers could not find, demonstrating its real-world applicability. We believe SbxBrk lays the groundwork for more rigorous testing of SFI mechanisms. We hope that future research will extend our methodology to other systems, such as Firefox's RLBox, and contribute to building more robust and secure software isolation.

Acknowledgments

We thank the anonymous reviewers for their valuable feedback. We are also grateful to Carl Smith for his comments on an earlier draft of this work. Finally, we thank Tobias Wienand for implementing FuzzilliSbx.

This work was funded by the European Research Council (ERC) under the consolidator grant RS^3 (101045669) and by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy (EXC 2092 CASA — 390781972). This material is based upon work supported by the National Science Foundation under Award No. 2232915 and by the Advanced Research Projects Agency for Health (ARPA-H) under Contract No. SP4701-23-C-0074. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation or ARPA-H.

References

[1] Armv8.5-A Memory Tagging Extension White Paper. https://developer.arm.com/documentation/102925/latest/, 2019. Accessed: September 10, 2025.

[2] Martín Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. Control-Flow Integrity Principles, Implementations, and Applications. *ACM Transactions on Information and System Security (TISSEC)*, 13(1), 2009.

[3] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. REDQUEEN: Fuzzing with Input-to-State Correspondence. In *Symposium on Network and Distributed System Security (NDSS)*, 2019.

[4] John Aycock. A Brief History of Just-In-Time. *ACM Computing Surveys (CSUR)*, 35(2):97–113, 2003.

[5] Nils Bars, Moritz Schloegel, Tobias Scharnowski, Nico Schiller, and Thorsten Holz. Fuzztruction: Using Fault Injection-based Fuzzing to Leverage Implicit Domain Knowledge. In *USENIX Security Symposium*, 2023.

[6] Nils Bars, Moritz Schloegel, Nico Schiller, Lukas Bernhard, and Thorsten Holz. No Peer, no Cry: Network Application Fuzzing via Fault Injection. In *ACM Conference on Computer and Communications Security (CCS)*, 2024.

[7] Lukas Bernhard, Tobias Scharnowski, Moritz Schloegel, Tim Blazytko, and Thorsten Holz. JIT-Picking: Differential Fuzzing of JavaScript Engines. In *ACM Conference on Computer and Communications Security (CCS)*, 2022.

[8] Lukas Bernhard, Nico Schiller, Moritz Schloegel, Nils Bars, and Thorsten Holz. DarthShader: Fuzzing WebGPU Shader Translators & Compilers. In *ACM Conference on Computer and Communications Security (CCS)*, 2024.

[9] bughunters.google.com. Chrome Vulnerability Reward Program Rules. https://bughunters.google.com/about/rules/chrome-friends/5745167867576320/chrome-vulnerability-reward-program-rules. Accessed: September 10, 2025.

[10] bugzilla.mozilla.org. Bugzilla: Evaluate libexpat CVE-2022-43680 Fix. https://bugzilla.mozilla.org/show_bug.cgi?id=CVE-2022-43680. Accessed: September 10, 2025.

[11] bugzilla.mozilla.org. Bugzilla: RLBox - Port libGraphite Usage Code to use the RLBox API. https://bugzilla.mozilla.org/show_bug.cgi?id=1566288. Accessed: September 10, 2025.

[12] bugzilla.mozilla.org. Bugzilla: RLBox - Port libOgg Usage Code in the OGGDemuxer to use the RLBox API. https://bugzilla.mozilla.org/show_bug.cgi?id=

1625876. Accessed: September 10, 2025.

[13] bugzilla.mozilla.org. Bugzilla: Sandbox libexpat using RLBox. https://bugzilla.mozilla.org/show_bug.cgi?id=1688452. Accessed: September 10, 2025.

[14] Nathan Burow, Scott A Carr, Joseph Nash, Per Larsen, Michael Franz, Stefan Brunthaler, and Mathias Payer. Control-Flow Integrity: Precision, Security, and Performance. *ACM Computing Surveys (CSUR)*, 50(1), 2017.

[15] chromium.googlesource.com. V8 Sandbox Readme. https://chromium.googlesource.com/v8/v8.git/+/refs/heads/main/src/sandbox/README.md. Accessed: September 10, 2025.

[16] chromium.org. V8 Bugtracker: Design Flaw in Synchronous Mojo Message Handling Introduces Unexpected Reentrancy and Allows for Multiple UAFs. https://issues.chromium.org/issues/40061398. Accessed: September 10, 2025.

[17] chromium.org. V8 Bugtracker: Incorrect Type Information on Math.expm1. https://issues.chromium.org/issues/40092352. Accessed: September 10, 2025.

[18] chromium.org. V8 Bugtracker: Mojo Message Validation Bypass due to Shared Memory. https://issues.chromium.org/issues/40063855. Accessed: September 10, 2025.

[19] chromium.org. V8 Bugtracker: Off by One in TurboFan Range Optimization for String.indexOf . https://issues.chromium.org/issues/40088942. Accessed: September 10, 2025.

[20] chromium.org. V8 Bugtracker: UaF in ImageCapture . https://issues.chromium.org/issues/40096129. Accessed: September 10, 2025.

[21] chromium.org. V8 Bugtracker: UAF in OfflinePageAutoFetcher::CancelSchedule . https://issues.chromium.org/issues/40095468. Accessed: September 10, 2025.

[22] John H Crawford and Patrick P Gelsinger. Programming the 80386 SYBEX. *Inc, San Francisco*, 1987.

[23] Daniele Cono D'Elia and Camil Demetrescu. On-stack replacement, distilled. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 166–180, 2018.

[24] Peter J Denning. Virtual Memory. *ACM Computing Surveys (CSUR)*, 2(3):153–189, 1970.

[25] docs.google.com. V8 Sandbox - High-Level Design Doc. https://docs.google.com/document/d/1FM4fQmIhEqPG8uGp5o9A-mnPB5BOeScZYpkHjo0KKA8/edit?usp=sharing. Accessed: September 10, 2025.

[26] Ulfar Erlingsson, Martín Abadi, Michael Vrable, Mihai Budiu, and George C Necula. XFI: Software Guards for System Address Spaces. In *Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

[27] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. AFL++: Combining Incremental Steps of Fuzzing Research. In *USENIX Workshop on Offensive Technologies (WOOT)*, 2020.

[28] Andrea Fioraldi, Dominik Christian Maier, Dongjia Zhang, and Davide Balzarotti. LibAFL: A Framework to Build Modular and Reusable Fuzzers. In *ACM Conference on Computer and Communications Security (CCS)*, 2022.

[29] Samuel Groß, Simon Koch, Lukas Bernhard, Thorsten Holz, and Martin Johns. FUZZILLI: Fuzzing for JavaScript JIT Compiler Vulnerabilities. In *Symposium on Network and Distributed System Security (NDSS)*, 2023.

[30] Jerome C. Huck and Jim Hays. Architectural Support For Translation Table Management In Large Address Space Machines. In *Annual International Symposium on Computer Architecture*, 1993.

[31] Zu-Ming Jiang, Jia-Ju Bai, Kangjie Lu, and Shi-Min Hu. Fuzzing Error Handling Code using Context-Sensitive Software Fault Injection. In *USENIX Security Symposium*, 2020.

[32] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. Evaluating Fuzz Testing. In *ACM Conference on Computer and Communications Security (CCS)*, 2018.

[33] Butler W Lampson. A Note on the Confinement Problem. *Communications of the ACM (CACM)*, 16(10), 1973.

[34] Peiyu Liu, Shouling Ji, Xuhong Zhang, Qinming Dai, Kangjie Lu, Lirong Fu, Wenzhi Chen, Peng Cheng, Wenhai Wang, and Raheem Beyah. IFIZZ: Deep-State and Efficient Fault-Scenario Generation to Test IoT Firmware. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2021.

[35] Stephen McCamant and Greg Morrisett. Efficient, Verifiable Binary Sandboxing for a CISC Architecture. Technical report, Massachusetts Institute of Technology, 2005.

[36] Greg Morrisett, Gang Tan, Joseph Tassarotti, Jean-Baptiste Tristan, and Edward Gan. RockSalt: Better, Faster, Stronger SFI for the x86. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2012.

[37] mozilla.org. JavaScript: Using Web Workers. https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API/Using_web_workers. Accessed: September 10, 2025.

[38] Shravan Narayan, Craig Disselkoen, Tal Garfinkel, Nathan Froyd, Eric Rahm, Sorin Lerner, Hovav Shacham, and Deian Stefan. Retrofitting Fine Grain Isolation in the Firefox Renderer. In *USENIX Security Symposium*, 2020.

[39] Soyeon Park, Sangho Lee, Wen Xu, HyunGon Moon, and Taesoo Kim. libmpk: Software Abstraction for Intel Memory Protection Keys (Intel MPK). In *USENIX Annual Technical Conference (ATC)*, 2019.

[40] Arvind S Raj, Wil Gibbs, Fangzhou Dong, Jayakrishna Menon Vadayath, Michael Tompkins, Steven Wirsz, Yibo Liu, Zhenghao Hu, Chang Zhu, Gokulkrishna Praveen Menon, Brendan Dolan-Gavitt, Adam Doupé, Ruoyu Wang, Yan Shoshitaishvili, and Tiffany Bao. Fuzz to the Future: Uncovering Occluded Future Vulnerabilities via Robust Fuzzing. In *ACM Conference on Computer and Communications Security (CCS)*, 2024.

[41] Andreas Rossberg. WebAssembly Core Specification.

[42] Tobias Scharnowski, Nils Bars, Moritz Schloegel, Eric Gustafson, Marius Muench, Giovanni Vigna, Christopher Kruegel, Thorsten Holz, and Ali Abbasi. Fuzzware: Using Precise MMIO Modeling for Effective Firmware Fuzzing. In *USENIX Security Symposium*, 2022.

[43] Moritz Schloegel, Nils Bars, Nico Schiller, Lukas Bernhard, Tobias Scharnowski, Addison Crump, Arash Ale-Ebrahim, Nicolai Bissantz, Marius Muench, and Thorsten Holz. SoK: Prudent Evaluation Practices for Fuzzing. In *IEEE Symposium on Security and Privacy (S&P)*, 2024.

[44] David Schrammel, Samuel Weiser, Stefan Steinegger, Martin Schwarzl, Michael Schwarz, Stefan Mangard, and Daniel Gruss. Donky: Domain Keys - Efficient In-Process Isolation for RISC-V and x86. In *USENIX Security Symposium*, 2020.

[45] Kostya Serebryany, Evgenii Stepanov, Aleksey Shlyapnikov, Vlad Tsyrklevich, and Dmitry Vyukov. Memory Tagging and How it Improves C/C++ Memory Safety, 2018.

[46] Shashank Sharma, Sai Ritvik Tanksalkar, Sourag Cherupattamoolayil, and Aravind Machiry. Fuzzing API Error Handling Behaviors using Coverage Guided Fault Injection. In *ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2024.

[47] v8.dev. The V8 Sandbox. https://v8.dev/blog/sandbox. Accessed: September 10, 2025.

[48] Robert Wahbe, Steven Lucco, Thomas E Anderson, and Susan L Graham. Efficient Software-based Fault Isolation. In *Symposium on Operating Systems Principles (SOSP)*, 1993.

[49] Robert N.M. Watson, Jonathan Woodruff, Peter G. Neumann, Simon W. Moore, Jonathan Anderson, David Chisnall, Nirav Dave, Brooks Davis, Khilan Gudka, Ben Laurie, Steven J. Murdoch, Robert Norton, Michael Roe, Stacey Son, and Munraj Vadera. CHERI: A Hybrid Capability-System Architecture for Scalable Software Compartmentalization. In *IEEE Symposium on Security and Privacy (S&P)*, 2015.

[50] Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. Native Client: A Sandbox for Portable, Untrusted x86 Native Code. In *IEEE Symposium on Security and Privacy (S&P)*, 2009.

Table 2: Different build flags used for building the V8 engine.

Flag	Description
`clang_use_chrome_plugins=false`	Disables Clang plugins that are shipped with V8 and improve its performance. Our custom Clang version may be incompatible with them.
`custom_toolchain` and `host_toolchain`	Allows us to use our custom Clang version.
`is_debug=false` and `dcheck_always_on=false`	Remove debug checks. Such checks would likely cause V8 to terminate before we can observe sandbox corruption.
`is_asan=true`	Enables ASan as an additional bug oracle.
`v8_enable_sandbox=true`	Enables the heap sandbox.
`v8_enable_memory_corruption_api=true`	Enables the memory-corruption API required to build proof-of-concepts (PoCs) for reporting bugs.
`v8_fuzzilli=true`	Enables fuzzing interface used by Fuzzilli (necessary for experiments where Fuzzilli is used).