

Mohabi: Disaggregating and Sandboxing the Firefox JavaScript Engine

Abhishek Sharma, Anand Balaji, Zachary Yedidia, Anthony Du, Taehyun Noh,
Iain Ireland, Jan de Mooij, Matthew Gaudet, Tal Garfinkel,
Deian Stefan, Hovav Shacham, Shravan Narayan

JavaScript Engines are particularly nice targets

ESET Research

RomCom exploits Firefox and Windows zero days in the wild

ESET Research details the analysis of a previously unknown vulnerability in Mozilla products exploited in the wild and another previously unknown Microsoft Windows vulnerability, combined in a zero-click exploit



Damien Schaeffer



Romain Dumont

26 Nov 2024 • 17 min. read

Operation Triangulation



CVE-2019-11707: Type confusion in Array.pop

Reporter Samuel Groß of Google Project Zero, Coinbase Security

Impact critical

Description

A type confusion vulnerability can occur when manipulating JavaScript objects due to issues in Array.pop. This can allow for an exploitable crash. We are aware of targeted attacks in the wild abusing this flaw.

References

[Bug 1544386](#)



Apple Patches Three Zero-day Vulnerabilities in WebKit Browser Engine (CVE-2023-32409, CVE-2023-28204, CVE-2023-32369, & CVE-2023-32373)

Posted by Diksha Ojha on May 19, 2023

Today, Google revealed that it patched the tenth Chrome zero-day exploited in the wild in 2024 by attackers or security researchers during hacking contests.

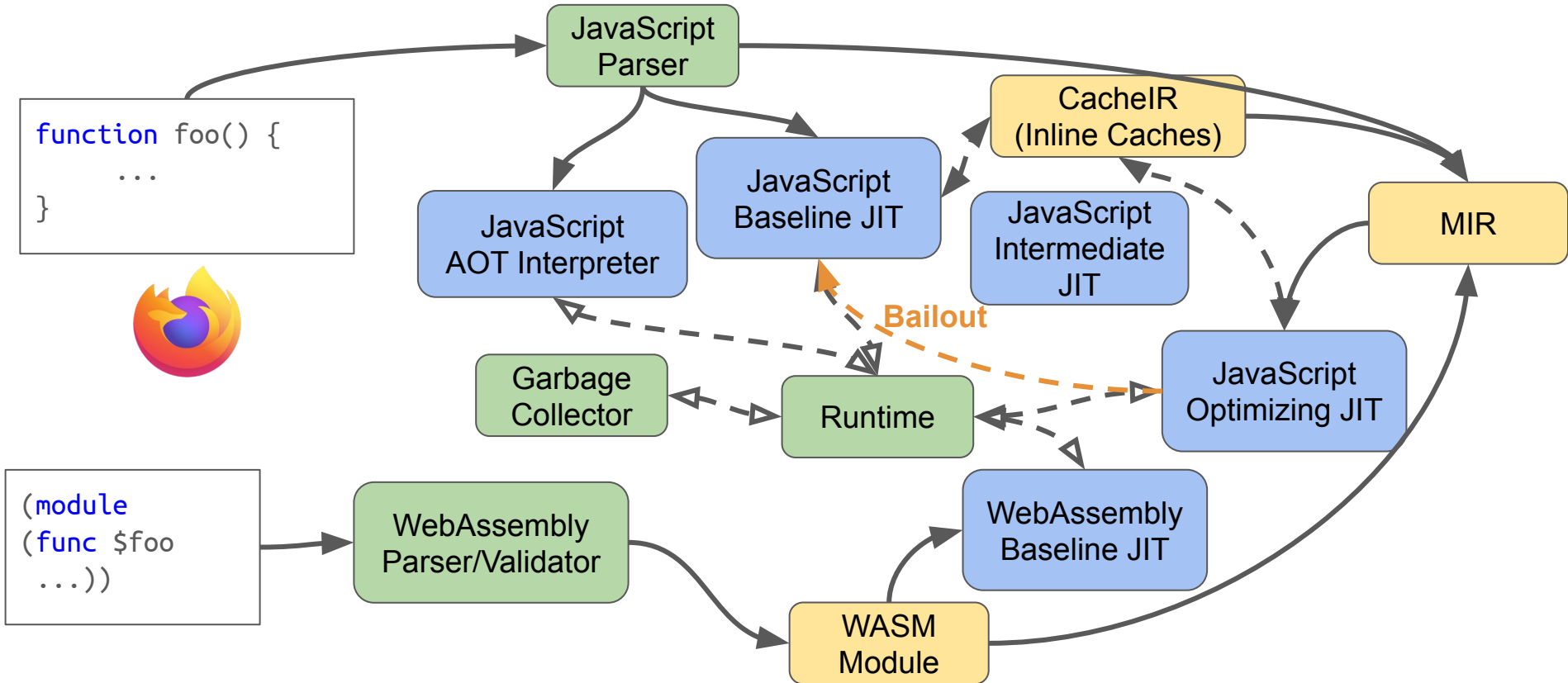
Tracked as [CVE-2024-7965](#) and reported by a security researcher known only as TheDog, the now-patched high-severity vulnerability is caused by a [bug](#) in the compiler backend when selecting the instructions to generate for just-in-time (JIT) compilation.

Google describes the vulnerability as an inappropriate implementation in Google Chrome's V8 JavaScript engine that can let remote attackers exploit heap corruption via a crafted HTML page.



Are seemingly an inexhaustible supply of memory-safety bugs!

Browsers' JavaScript engines are complex!



Bugs in such complex systems are inevitable!

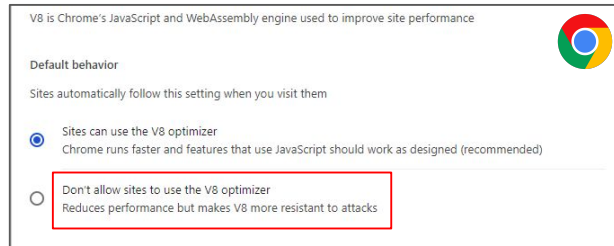
Browser vendor response: Best-effort mitigations

Testing: Static analysis, Fuzzing, LLMs

- Identifies many bugs, but no concrete guarantees

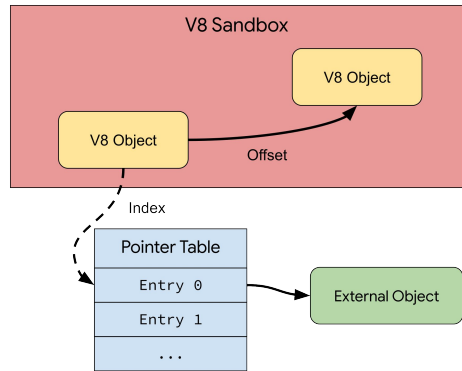
Disable JavaScript JIT optimizer:

- ~40% of 2025 (Chrome) exploits didn't rely on JIT



Ubercage: Chrome's partial JavaScript sandbox (~2021)

- Sandboxes JIT heap + some JS runtime heap access
- Strongest mitigation to date, very fast
- **Still best effort...** Over 150 bypasses so far



Can we design a more principled sandbox for
the JavaScript Engine?

Mohabi in a nutshell: isolate JS engines using in-process sandboxing

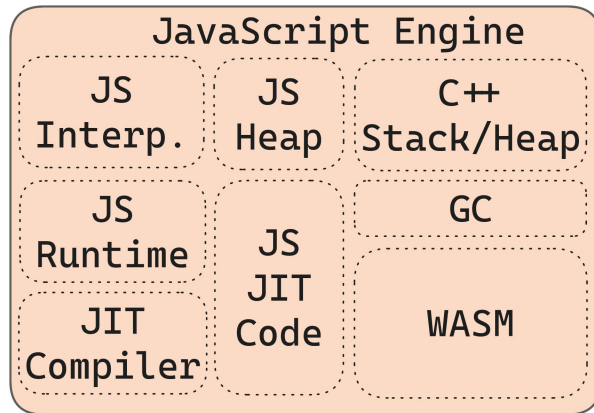
Pushes the limits of in-process sandboxing!

Challenge: Scale techniques to this use case?

Our approach:

1. Techniques for semi-automated disaggregation
 - Disaggregation: Separate the data & code that must be sandboxed
 - The key is picking the boundary carefully
2. A new sandbox tool (MH-LFI) that can isolate JS engines
 - Can sandbox JIT compilers & supports large memory footprints

⇒ **Practical approach to securing Firefox with modest overheads.**



Outline

1. Mohabi: Disaggregate the JavaScript Engine from the Browser
2. MH-LFI: Scaling Sandboxing to the JavaScript Engine
3. Evaluating Mohabi's Performance and Security

How can we disaggregate the JavaScript engine?

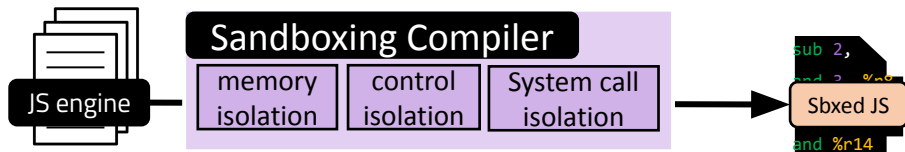
Step 1: Assume a deny-by-default memory sharing policy

- JS engine cannot modify data allocated by the browser
- JS engine cannot invoke browser functions

Step 2: Permit controlled data and control-flow sharing through wrapper types

- Wrapper types communicate with a sandbox runtime to permit secure sharing

Step 3: Isolate the JavaScript Engine with a sandboxing compiler



What does this look like?

For a deny-by-default memory sharing policy, we must:

Ensure all writes from JS engines are to its own allocations

- Browser allocs buffer of type T → passes it to the JS engine, and the engine writes to it
- In this situation, we must change the browser code to use `SandboxHeap<T>`
- `SandboxHeap<T>` → new Mozilla wrapper type that permit this
 - Works by moving the allocation to sandbox-accessible memory

Similarly, we have wrapper types for:

- Shared stack memory: `SandboxStack<T>`
- Firefox functions marked callable by the JS engine: `SandboxCallback<T>`

What does this look like?

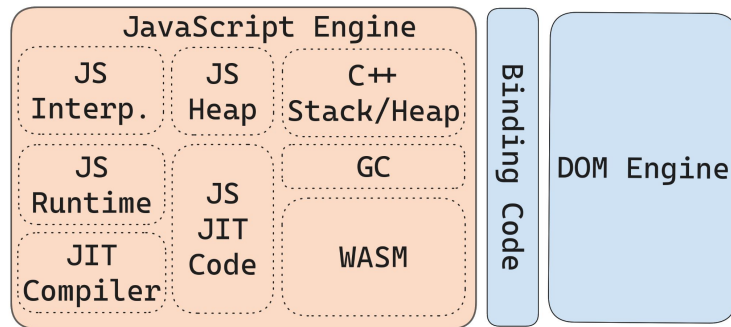
The full JS engine ↔ DOM integration

- 20,000 shared data-structures
- 15,000 function calls.

All in the “browser binding layer”

- Glue code between JS engine & DOM
- Too large to modify by hand

Browsers already auto-generate this code!



Can we modify the code-generator to use wrapper types?

Challenge: Complex data/control sharing patterns that make this tricky.

For example, we need to handle:

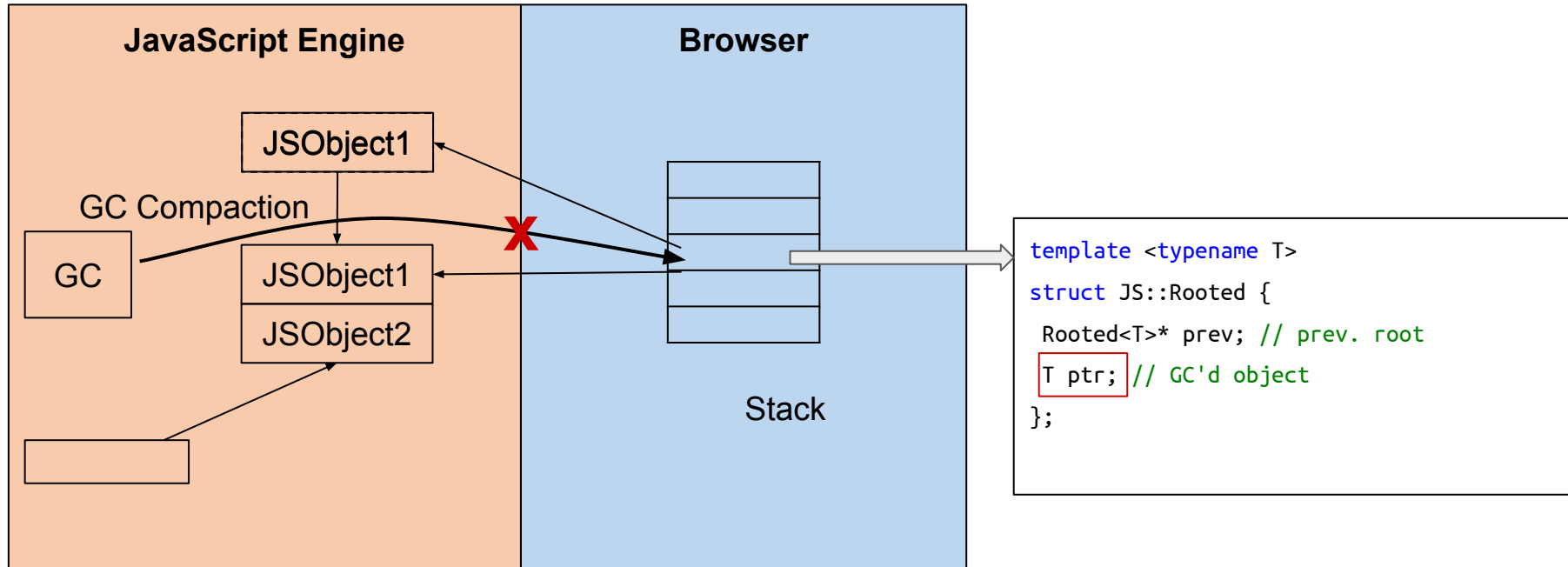
Structs where some fields are shared

Objects in the sandbox that inherit virtual functions defined outside the sandbox

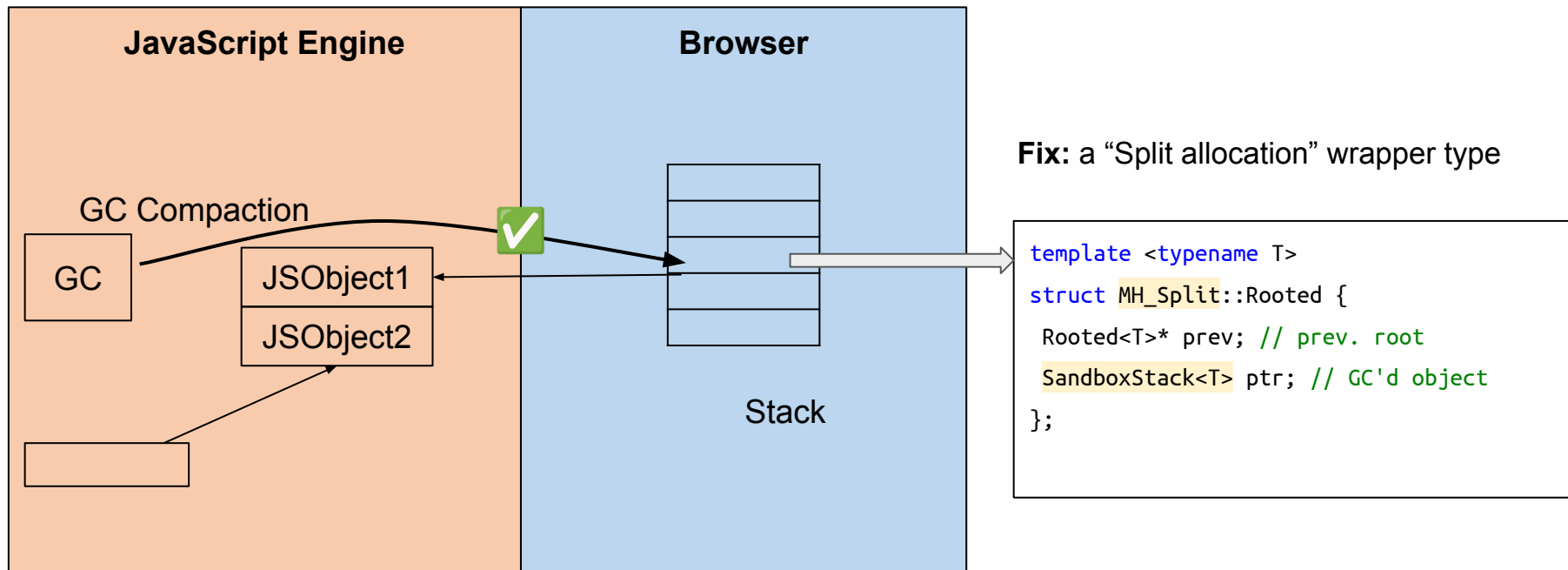
Objects in the sandbox where members interact with browser binding code

Applications pointers that are stored in sandbox memory

Sandboxing breaks Garbage Collection



Sandboxing breaks Garbage Collection



Replace all uses of `JS::Rooted<T>` in the binding layer

Only a few `JS::Rooted<T>` $\Rightarrow$ `MHSplit::Rooted<T>`

- Huge payoff $\Rightarrow$ Applied at 1,000's of sites

The other wrapper types had similar benefits

- Sandbox Reflectors: 15,000 callbacks
- Stub generators: 2,250 functions
- Inheritance (Trampoline) Types: ~20 hierarchies

Takeaway: Binding layer is well-defined chokepoint for disaggregation

Pick a different boundary? E.g., sandbox JITs like Ubercage

No automation, manual modifications

How can we disaggregate the JavaScript engine?

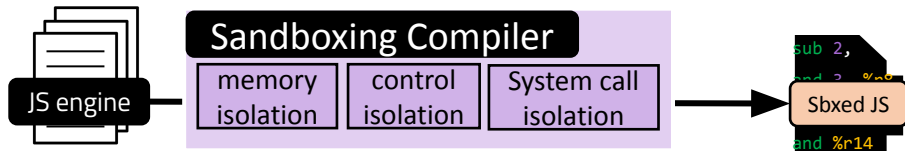
Step 1: Assume a deny-by-default memory sharing policy

- JS engine cannot modify data allocated by the browser
- JS engine cannot invoke browser functions

Step 2: Permit controlled data and control-flow sharing through wrapper types

- Wrapper types communicate with a sandbox runtime to permit secure sharing

Step 3: Isolate the JavaScript Engine with a sandboxing compiler



Outline

1. Mohabi: Disaggregate the JavaScript Engine from the Browser
2. MH-LFI: Scaling Sandboxing to the JavaScript Engine
3. Evaluating Mohabi's Performance and Security

Why build a new sandboxing toolchain?

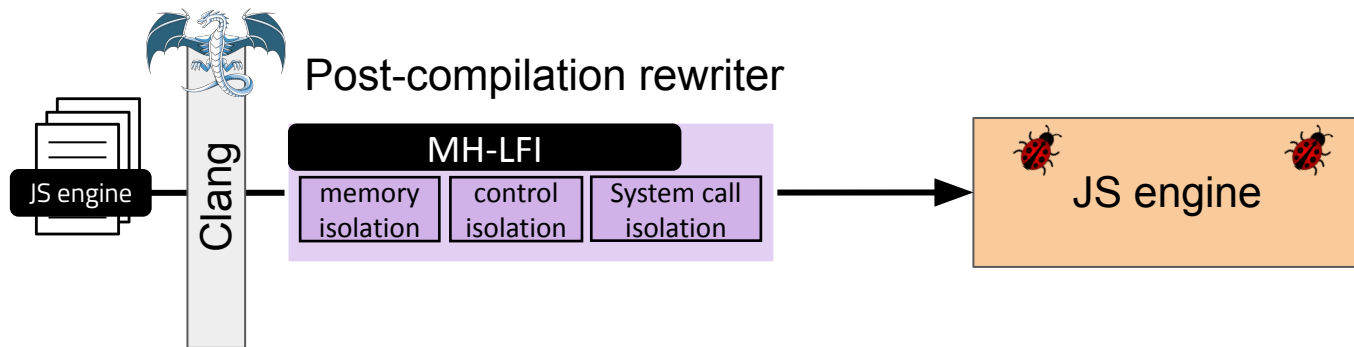
Numerous sandboxing toolchains available today!

- WebAssembly, Native Client etc.

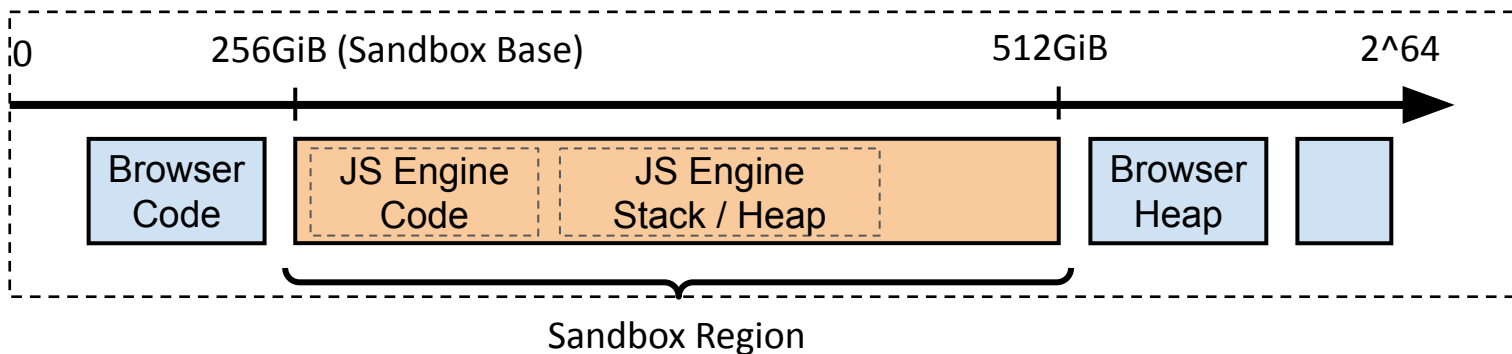
No toolchain supports the combination of features we need

- Sandboxes with large memory
- Sandbox a JIT compilation
- Modern compiler toolchain
- No hardware support

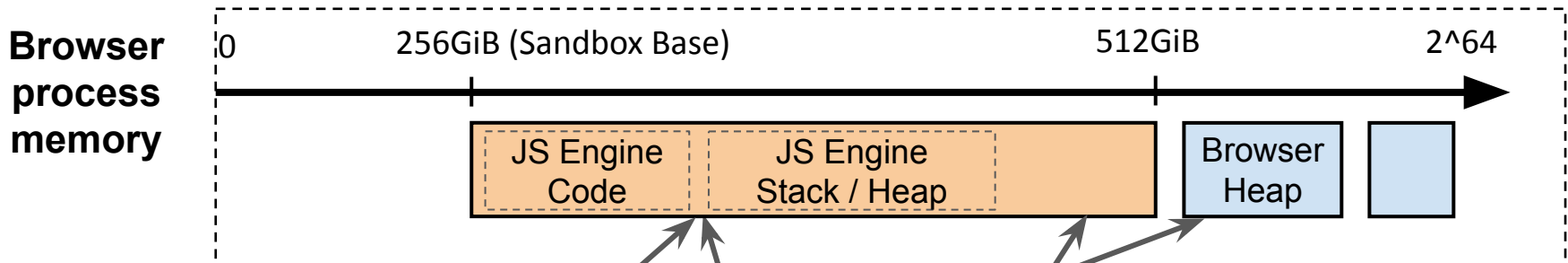
MH-LFI: A scalable approach to software sandboxing



**Browser
process
memory**



How does the MH-LFI rewriter work?

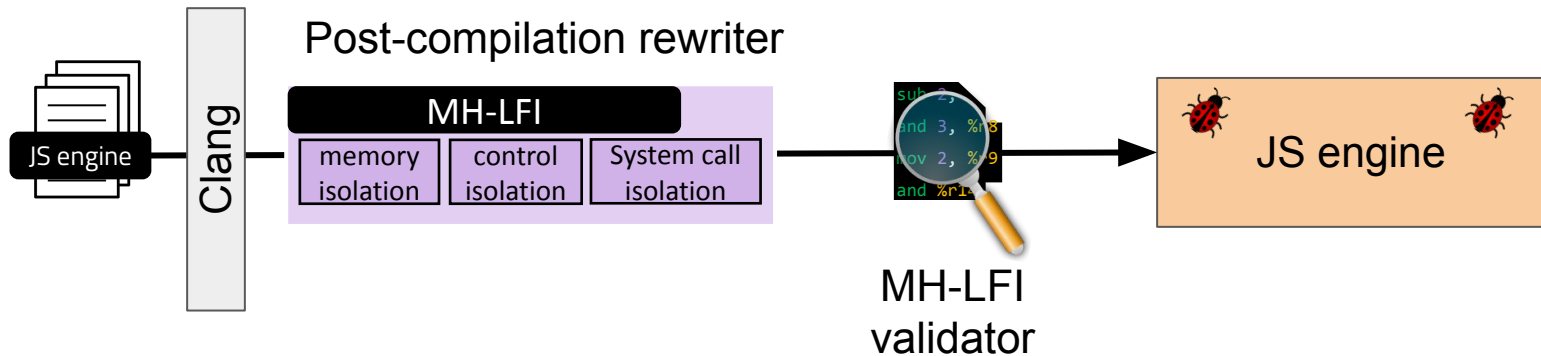


Original Code	Rewritten assembly
<pre>movq %rdi, (%rax) // (uint64_t*) rax = rdi</pre>	<pre>movq %rax, %r11 andq %r15, %r11 movq %rdi, (%r14, %r11)</pre>

Takeaway: With care, this transformation can be done efficiently in the external rewriter.

(Details of optimizations in paper)

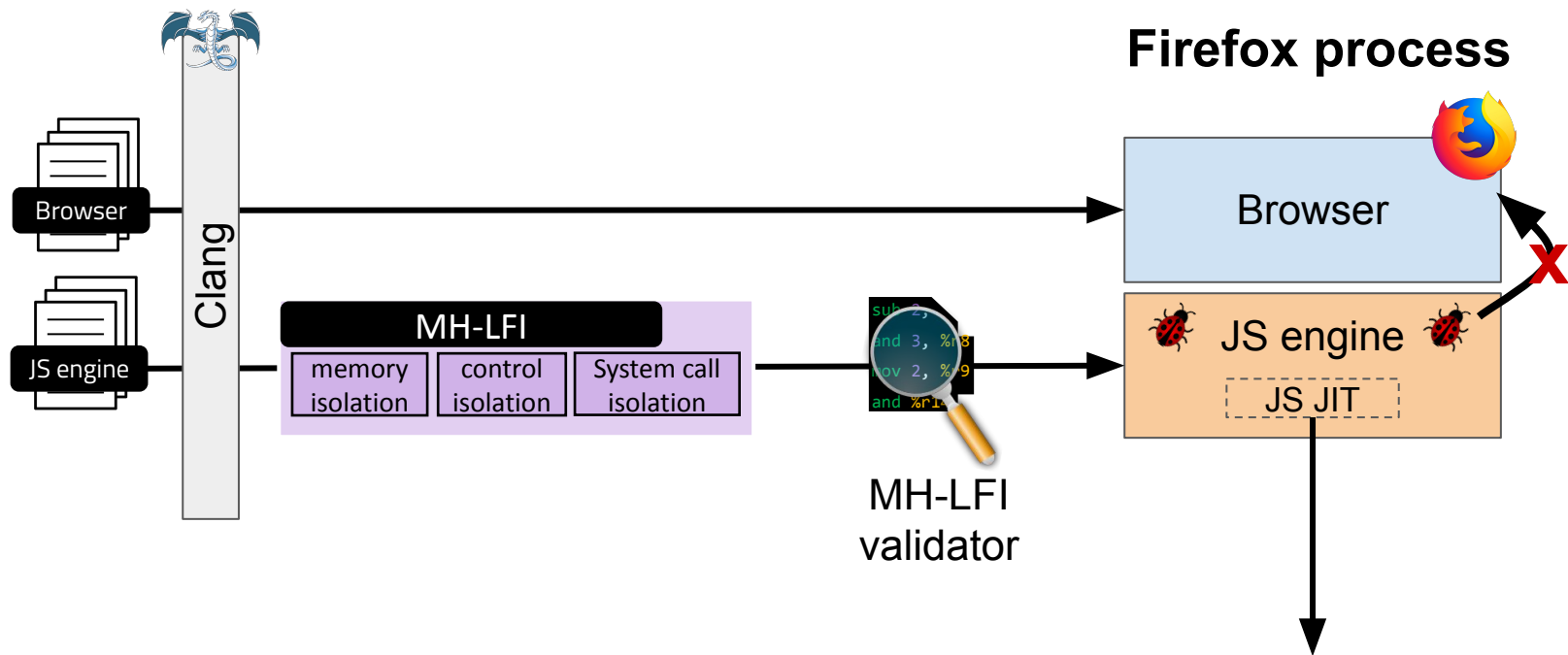
Can we remove the rewriter from the TCB?



We built a small validator that looks for the required rewrites in binaries

Key: Validator is small (~1000 LOC), rewriter can be clever (uses optimizations)

Building the JavaScript Engine with MH-LFI



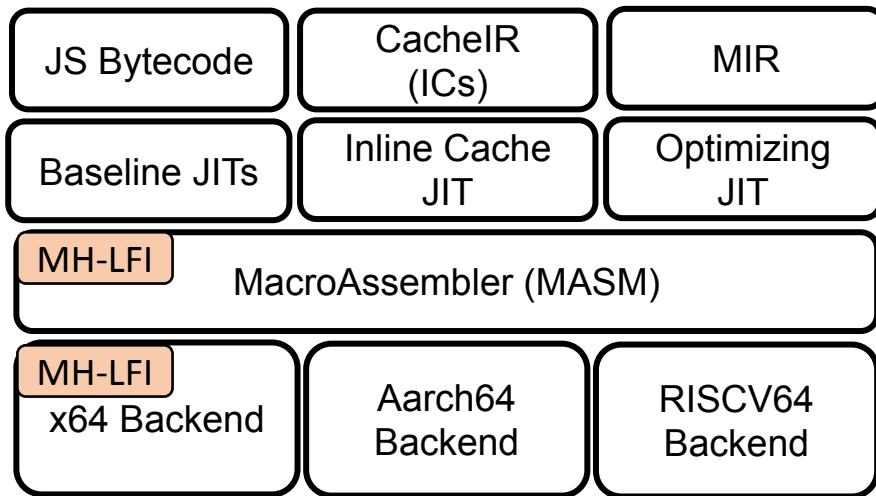
JavaScript JIT compilers need to perform the same rewrites

- How do we modify the JS engine's many JIT compilers?

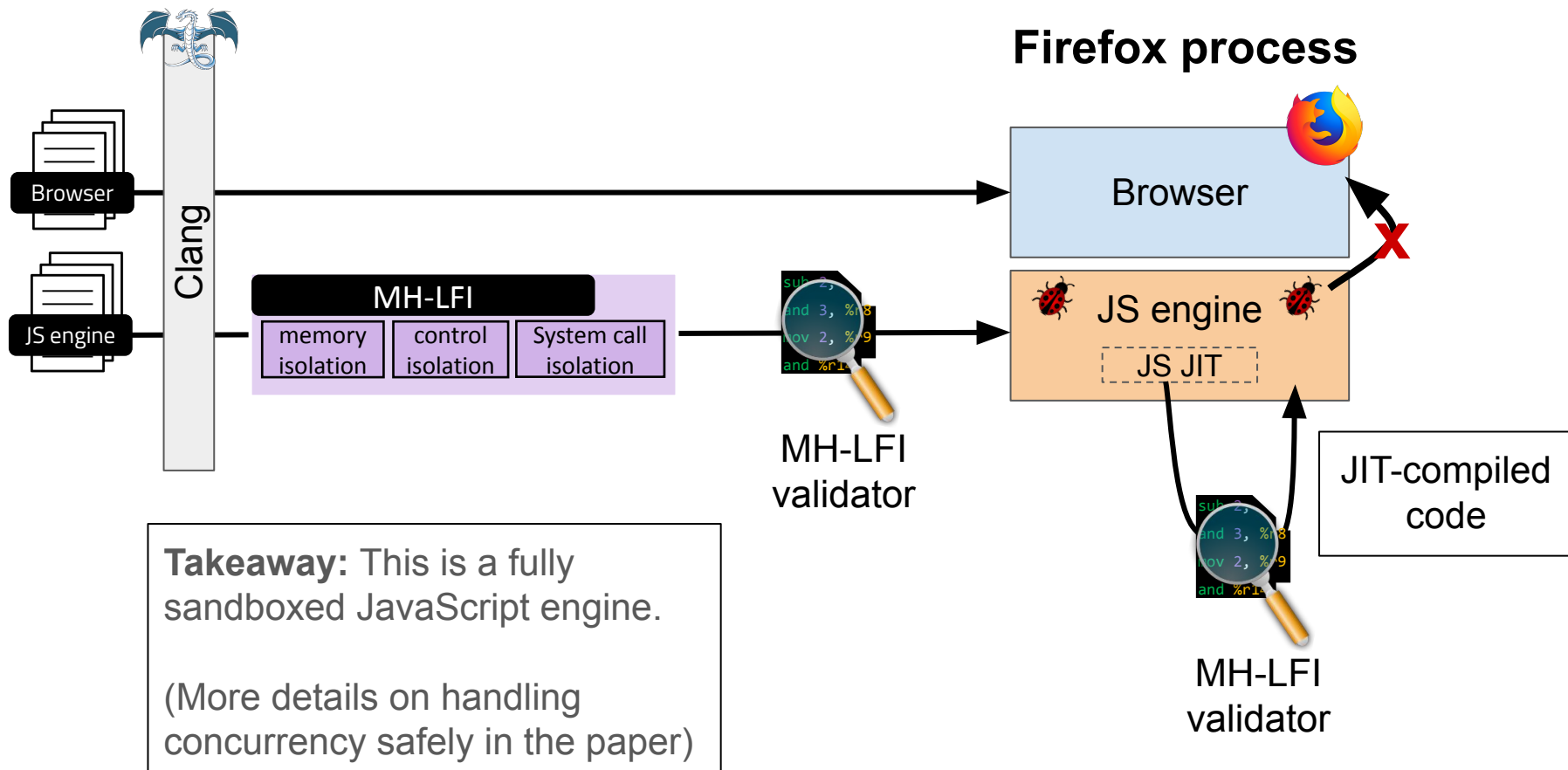
Surprisingly easy to modifying all JIT compilers at once!

Observation: Firefox's JITs have a common layer called MacroAssembler

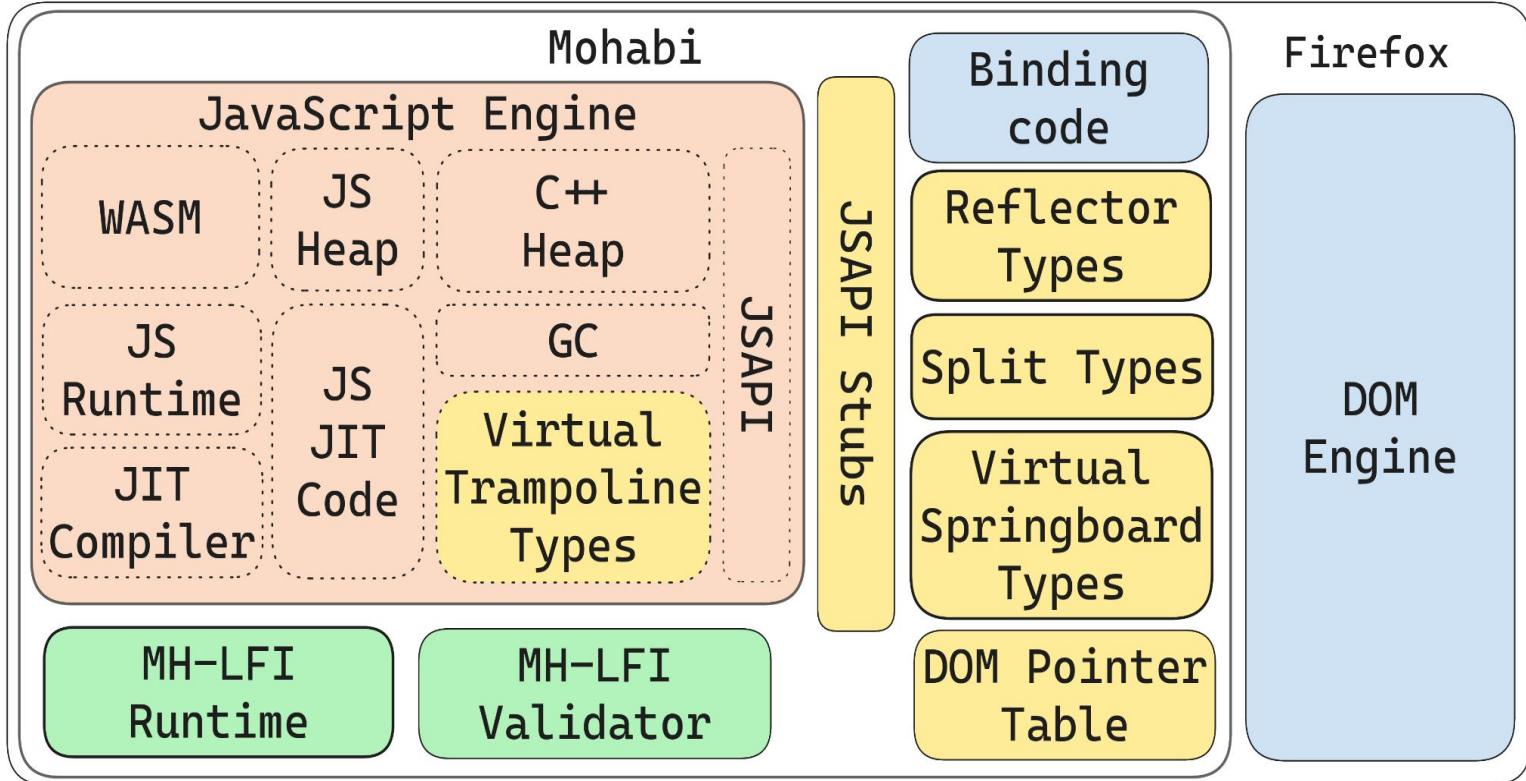
- Common browser pattern. E.g., also in Chrome
- Modify the JITs to do the same assembly rewrites as MH-LFI



JIT compilers now generate safely sandboxed code



The complete Mohabi system



Outline

1. Mohabi: Disaggregate the JavaScript Engine from the Browser
2. MH-LFI: Scaling Sandboxing to the JavaScript Engine
3. Evaluating Mohabi's Performance and Security

Evaluating Mohabi's Performance

Speedometer 3.1

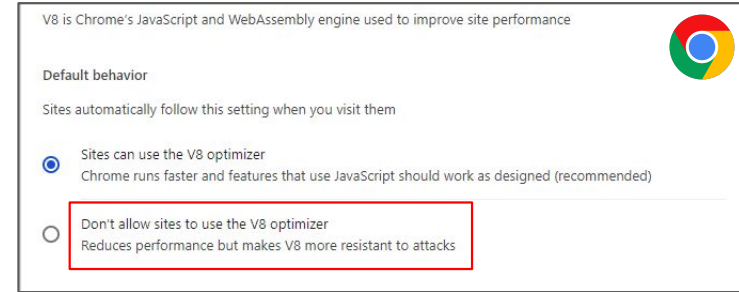
24.43%

JetStream 2.2

24.82%

Alternative: Disable JITs like Chrome/Edge/Safari

- **Speedometer:** 53%
- **JetStream:** Firefox can't run this w/o the JIT
- **Not secure:** wouldn't block 40% of 2025 attacks



Evaluating Mohabi's Security

JavaScript exploit primitives can't break Mohabi

- Code pointer hijacking ✓
- Overwriting Exec. Memory ✓
- JIT spray attacks ✓
- Corruption via concurrency ✓
- Unsafe syscalls ✓

After we completed Mohabi: Mozilla announced that the Mythos LLM had discovered numerous bugs

Behind the Scenes Hardening Firefox with Claude Mythos Preview



By [Brian Grinstead](#), [Christian Holler](#), [Frederik Braun](#)

Posted on [May 7, 2026](#) in [Featured Article](#), [Firefox](#), and [Security](#)

Mohabi also mitigates multiple recent bugs found by Mythos

Summary

Mohabi: Isolates the complex JavaScript engines in a modern browser

To our knowledge, the most ambitious use of in-process sandboxing

Takeaways: Picking the right layer matters for scalable disaggregation/sandboxing

1. Browser binding layer $\Rightarrow$ chokepoint for semi-automated disaggregation
2. MacroAssembler layer $\Rightarrow$ centrally modify JITs to produce sandboxed code
3. External rewriter $\Rightarrow$ simplifies producing scalable, up-to-date sandboxing tools
 - Can support complex features like large memory and sandboxed JITs

$\Rightarrow$ Mohabi shows that we can secure JavaScript engines at very reasonable overheads